Developing Database Applications Using MySQL Connector/C++

by Giri Mandalika, Sun Microsystems Sofware Engineer

This tutorial will show you the essential steps to build and install MySQL Connector/C++ driver, with simple examples to connect, insert, and retrieve data from a MySQL database. Because the focus is on database connectivity from a C++ application, this document assumes that some kind of MySQL database is already up and accessible from the client machine.

Application developers who are new to MySQL Connector/C++ but not to C++ programming and MySQL database, are the target audience of this tutorial.

Listed below are the tools and technologies used to compile, build and run the examples in this tutorial.

Database	MySQL Server 5.1.24-rc
C++ Driver	MySQL Connector/C++ 1.0.5
MySQL Client Library	MySQL Connector/C 6.0
Compiler	Sun Studio 12 C++ compiler
Make	CMake 2.6.3
Operating System	OpenSolaris 2008.11 32-bit
CPU / ISA	Intel Centrino / x86
Hardware	Toshiba Tecra M2 Laptop

Contents

MySQL C++ Driver Based on JDBC 4.0 Specification

MySQL Connector/C++ is one of the latest connectors for MySQL, developed by Sun Microsystems. The MySQL connector for C++ provides an object-oriented application programming interface (API) and a database driver for connecting C++ applications to the MySQL Server.

The development of Connector/C++ took a different approach compared to the existing drivers for C++ by implementing the JDBC API in C++ world. In other words, Connector/C++ driver's interface is mostly based on Java programming language's JDBC API. Java Database Connectivity (JDBC) API is the industry standard for connectivity between the Java programming language and a wide range of databases. Connector/C++ implemented a significant percentage of the JDBC 4.0 specification. C++ application developers who are familiar with JDBC programming may find this useful, and as a result, it could improve application development time.

The following classes were implemented by the MySQL Connector/C++:

- Driver
- Connection
- Statement
- PreparedStatement
- ResultSet
- Savepoint
- DatabaseMetaData
- ResultSetMetaData
- ParameterMetaData

The Connector/C++ driver can be used to connect to MySQL 5.1 and later versions.

Prior to MySQL Connector/C++, C++ application developers were required to use either the non-standard & procedural MySQL C API directly or the MySQL++ API, which is a C++ wrapper for the MySQL C API.

Installing MySQL Connector/C++

Binary Installation

Starting with release 1.0.4, Connector/C++ is available in binary form for Solaris, Linux, Windows, FreeBSD, Mac OS X, HP-UX and AIX platforms. MSI installer and a binary zip file without the installer is available for Windows, where as the binary package is available as compressed GNU TAR archive (tar.gz) for the rest of the platforms. You can download the pre-compiled binary from the Connector/C++ download page.

Installation from the binary package is very straight forward on Windows and other platforms - simply unpacking the archive in a desired location installs the Connector/C++ driver. Both statically linked and the dynamically linked Connector/C++ driver can be found in `lib` directory under the driver installation directory. If you plan to use the dynamically linked version of MySQL Connector/C++, ensure that the runtime linker can find the MySQL Client Library. Consult your operating system documentation for the steps to modify and expand the search path for libraries. In case you cannot modify the library search path, copy your application, the MySQL Connector/C++ driver and the MySQL Client Library into the same directory. This approach may work on most of the platforms as they search the originating directory by default, before searching for the required dynamic libraries elsewhere.

Source Installation

Those who want to build the connector driver from the source code, please check the Installing MySQL Connector/C++ from Source section in Appendix I for detailed instructions.

Runtime Dependencies

Because the Connector/C++ driver is linked against the MySQL Client Library, dynamically linked C++ applications that use Connector/C++ will require the MySQL client programming support installed along with the connector driver, on the machines where the application is supposed to run. Static linking with the MySQL Client Library is one of the options to eliminate this runtime dependency. However due to various reasons, static linking is discouraged in building larger applications.

IDE for Developing C++ Applications

If you are looking for an integrated development environment (IDE) to develop C/C++ applications, consider using the free and open NetBeans platform. NetBeans C/C++ Development Pack lets C/C++ developers use their specified set of compilers and tools in conjunction with NetBeans IDE to build native applications on Solaris, Linux, Windows and Mac OS X. The C/C++ development pack makes the editor language-aware for C/C++ and provides project templates, a dynamic class browser, Makefile support & debugger functionality. It is possible to extend C/C++ development pack base functionality with new features, modules and plug-ins.

MySQL Connector/C++: How to Build a Client using NetBeans 6.5 (for Dummies) tutorial has instructions to use NetBeans IDE to build client applications based on Connector/C++. In addition to the above tutorial, Installing and Configuring C/C++ Support tutorial on NetBeans.org web site will help you with the installation and configuration steps for the NetBeans C/C++ development pack, and Getting Started With the NetBeans C/C++ Development Pack tutorial provides the basic steps involved in developing a C/C++ application using the NetBeans C/C++ Development Pack.

Create the City Table in the test Database for Code Examples

The code samples in this tutorial try to retrieve the data from the `City` table in the MySQL `test` database. The table structure and the data from the `City` table are shown here by using the `mysql` client. The MySQL Server is running on the default port 3306.

```
# mysql -u root -p
Enter password: admin
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5
Server version: 5.1.24-rc-standard Source distribution

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE test;
Database changed

mysql> DESCRIBE City;
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| CityName | varchar(30) | YES  |     |  NULL   |       |
+----------+-------------+------+-----+---------+-------+
1 row in set (0.07 sec)
```

```
mysql> SHOW CREATE TABLE City\G
*************************** 1. row ***************************
       Table: City
Create Table: CREATE TABLE `City` (
  `CityName` varchar(30) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=ascii
1 row in set (0.00 sec)

mysql> SELECT * FROM City;

+--------------------+
| CityName           |
+--------------------+
| Hyderabad, India   |
| San Francisco, USA |
| Sydney, Australia  |
+--------------------+
3 rows in set (0.17 sec)
```

`bash` shell was used to show all the examples in this tutorial.

Testing the MySQL Database Connectivity With the Connector/C++

The following C++ code sample demonstrates how to connect to a MySQL Server running on the same host, using the MySQL Connector for C++. The code sample connects to the MySQL database `test` by using the JDBC like API provided by the Connector C++, executes a query to retrieve all the rows from the table `City`, extracts the data from the result set and displays it on the standard output, inserts couple of rows of data into the table `City` using the Prepared Statements, demonstrates transactions using savepoints and examines the result set and database metadata.

The sample code is provided only for the purpose of demonstration. It does not recommend the readers to adopt a particular style of coding. To keep it simple, the sample code assumes that the user always provides well-formed input - hence there is no explicit error checking code in the following example. Use discretion in re-using the sample code.

```
# cat MySQLConnectorC++Client.cpp

/* Standard C++ headers */
#include <iostream>
#include <sstream>
#include <memory>
#include <string>
#include <stdexcept>

/* MySQL Connector/C++ specific headers */
#include <driver.h>
#include <connection.h>
#include <statement.h>
#include <prepared_statement.h>
#include <resultset.h>
#include <metadata.h>
#include <resultset_metadata.h>
#include <exception.h>
#include <warning.h>

#define DBHOST "tcp://127.0.0.1:3306"
#define USER "root"
#define PASSWORD "admin"
```

```
#define DATABASE "test"

#define NUMOFFSET 100
#define COLNAME 200

using namespace std;
using namespace sql;

static void retrieve_data_and_print (ResultSet *rs, int type, int colidx, string colname) {

        /* retrieve the row count in the result set */
        cout << "\nRetrieved " << rs -> rowsCount() << " row(s)." << endl;

        cout << "\nCityName" << endl;
        cout << "--------" << endl;

        /* fetch the data : retrieve all the rows in the result set */
        while (rs->next()) {
                if (type == NUMOFFSET) {
                        cout << rs -> getString(colidx) << endl;
                } else if (type == COLNAME) {
                        cout << rs -> getString(colname) << endl;
                } // if-else
        } // while

        cout << endl;

} // retrieve_data_and_print()

static void retrieve_dbmetadata_and_print (Connection *dbcon) {

        if (dbcon -> isClosed()) {
                throw runtime_error("DatabaseMetaData FAILURE - database connection closed");
        }

        cout << "\nDatabase Metadata" << endl;
        cout << "-----------------" << endl;

        cout << boolalpha;

        /* The following commented statement won't work with Connector/C++ 1.0.5 and later */
        //auto_ptr < DatabaseMetaData > dbcon_meta (dbcon -> getMetaData());

        DatabaseMetaData *dbcon_meta = dbcon -> getMetaData();

        cout << "Database Product Name: " << dbcon_meta -> getDatabaseProductName() << endl;
        cout << "Database Product Version: " << dbcon_meta -> getDatabaseProductVersion() << endl;
        cout << "Database User Name: " << dbcon_meta -> getUserName() << endl << endl;

        cout << "Driver name: " << dbcon_meta -> getDriverName() << endl;
        cout << "Driver version: " << dbcon_meta -> getDriverVersion() << endl << endl;

        cout << "Database in Read-Only Mode?: " << dbcon_meta -> isReadOnly() << endl;
        cout << "Supports Transactions?: " << dbcon_meta -> supportsTransactions() << endl;
        cout << "Supports DML Transactions only?: " << dbcon_meta ->
                 supportsDataManipulationTransactionsOnly() << endl;
        cout << "Supports Batch Updates?: " << dbcon_meta -> supportsBatchUpdates() << endl;
        cout << "Supports Outer Joins?: " << dbcon_meta -> supportsOuterJoins() << endl;
        cout << "Supports Multiple Transactions?: " << dbcon_meta ->
                 supportsMultipleTransactions() << endl;
        cout << "Supports Named Parameters?: " << dbcon_meta -> supportsNamedParameters() << endl;
        cout << "Supports Statement Pooling?: " << dbcon_meta -> supportsStatementPooling() << endl;
        cout << "Supports Stored Procedures?: " << dbcon_meta -> supportsStoredProcedures() << endl;
        cout << "Supports Union?: " << dbcon_meta -> supportsUnion() << endl << endl;
```

```
        cout << "Maximum Connections: " << dbcon_meta -> getMaxConnections() << endl;
        cout << "Maximum Columns per Table: " << dbcon_meta -> getMaxColumnsInTable() << endl;
        cout << "Maximum Columns per Index: " << dbcon_meta -> getMaxColumnsInIndex() << endl;
        cout << "Maximum Row Size per Table: " << dbcon_meta -> getMaxRowSize() << " bytes" << endl;

        cout << "\nDatabase schemas: " << endl;

        auto_ptr < ResultSet > rs ( dbcon_meta -> getSchemas());

        cout << "\nTotal number of schemas = " << rs -> rowsCount() << endl;
        cout << endl;

        int row = 1;

        while (rs -> next()) {
               cout << "\t" << row << ". " << rs -> getString("TABLE_SCHEM") << endl;
               ++row;
        } // while

        cout << endl << endl;

} // retrieve_dbmetadata_and_print()

static void retrieve_rsmetadata_and_print (ResultSet *rs) {

        if (rs -> rowsCount() == 0) {
               throw runtime_error("ResultSetMetaData FAILURE - no records in the result set");
        }

        cout << "ResultSet Metadata" << endl;
        cout << "------------------" << endl;

        /* The following commented statement won't work with Connector/C++ 1.0.5 and later */
        //auto_ptr < ResultSetMetaData > res_meta ( rs -> getMetaData() );

        ResultSetMetaData *res_meta = rs -> getMetaData();

        int numcols = res_meta -> getColumnCount();
        cout << "\nNumber of columns in the result set = " << numcols << endl << endl;

        cout.width(20);
        cout << "Column Name/Label";
        cout.width(20);
        cout << "Column Type";
        cout.width(20);
        cout << "Column Size" << endl;

        for (int i = 0; i < numcols; ++i) {
               cout.width(20);
               cout << res_meta -> getColumnLabel (i+1);
               cout.width(20);
               cout << res_meta -> getColumnTypeName (i+1);
               cout.width(20);
               cout << res_meta -> getColumnDisplaySize (i+1) << endl << endl;
        }

        cout << "\nColumn \"" << res_meta -> getColumnLabel(1);
        cout << "\" belongs to the Table: \"" << res_meta -> getTableName(1);
        cout << "\" which belongs to the Schema: \"" << res_meta -> getSchemaName(1) << "\"" <<
               endl << endl;

} // retrieve_rsmetadata_and_print()

int main(int argc, const char *argv[]) {

        Driver *driver;
```

```
Connection *con;
Statement *stmt;
ResultSet *res;
PreparedStatement *prep_stmt;
Savepoint *savept;

int updatecount = 0;

/* initiate url, user, password and database variables */
string url(argc >= 2 ? argv[1] : DBHOST);
const string user(argc >= 3 ? argv[2] : USER);
const string password(argc >= 4 ? argv[3] : PASSWORD);
const string database(argc >= 5 ? argv[4] : DATABASE);

try {
        driver = get_driver_instance();

        /* create a database connection using the Driver */
        con = driver -> connect(url, user, password);

        /* alternate syntax using auto_ptr to create the db connection */
        //auto_ptr  con (driver -> connect(url, user, password));

        /* turn off the autocommit */
        con -> setAutoCommit(0);

        cout << "\nDatabase connection\'s autocommit mode = " << con -> getAutoCommit() << endl;

        /* select appropriate database schema */
        con -> setSchema(database);

        /* retrieve and display the database metadata */
        retrieve_dbmetadata_and_print (con);

        /* create a statement object */
        stmt = con -> createStatement();

        cout << "Executing the Query: \"SELECT * FROM City\" .." << endl;

        /* run a query which returns exactly one result set */
        res = stmt -> executeQuery ("SELECT * FROM City");

        cout << "Retrieving the result set .." << endl;

        /* retrieve the data from the result set and display on stdout */
        retrieve_data_and_print (res, NUMOFFSET, 1, string("CityName"));

        /* retrieve and display the result set metadata */
        retrieve_rsmetadata_and_print (res);

        cout << "Demonstrating Prepared Statements .. " << endl << endl;

        /* insert couple of rows of data into City table using Prepared Statements */
        prep_stmt = con -> prepareStatement ("INSERT INTO City (CityName) VALUES (?)");

        cout << "\tInserting \"London, UK\" into the table, City .." << endl;

        prep_stmt -> setString (1, "London, UK");
        updatecount = prep_stmt -> executeUpdate();

        cout << "\tCreating a save point \"SAVEPT1\" .." << endl;
        savept = con -> setSavepoint ("SAVEPT1");

        cout << "\tInserting \"Paris, France\" into the table, City .." << endl;
```

```
                prep_stmt -> setString (1, "Paris, France");
                updatecount = prep_stmt -> executeUpdate();

                cout << "\tRolling back until the last save point \"SAVEPT1\" .." << endl;
                con -> rollback (savept);
                con -> releaseSavepoint (savept);

                cout << "\tCommitting outstanding updates to the database .." << endl;
                con -> commit();

                cout << "\nQuerying the City table again .." << endl;

                /* re-use result set object */
                res = NULL;
                res = stmt -> executeQuery ("SELECT * FROM City");

                /* retrieve the data from the result set and display on stdout */
                retrieve_data_and_print (res, COLNAME, 1, string ("CityName"));

                cout << "Cleaning up the resources .." << endl;

                /* Clean up */
                delete res;
                delete stmt;
                delete prep_stmt;
                con -> close();
                delete con;

        } catch (SQLException &e) {
                cout << "ERROR: SQLException in " << __FILE__;
                cout << " (" << __func__<< ") on line " << __LINE__ << endl;
                cout << "ERROR: " << e.what();
                cout << " (MySQL error code: " << e.getErrorCode();
                cout << ", SQLState: " << e.getSQLState() << ")" << endl;

                if (e.getErrorCode() == 1047) {
                        /*
                        Error: 1047 SQLSTATE: 08S01 (ER_UNKNOWN_COM_ERROR)
                        Message: Unknown command
                        */
                        cout << "\nYour server does not seem to support Prepared Statements at all. ";
                        cout << "Perhaps MYSQL < 4.1?" << endl;
                }

                return EXIT_FAILURE;
        } catch (std::runtime_error &e) {

                cout << "ERROR: runtime_error in " << __FILE__;
                cout << " (" << __func__ << ") on line " << __LINE__ << endl;
                cout << "ERROR: " << e.what() << endl;

                return EXIT_FAILURE;
        }

        return EXIT_SUCCESS;
} // main()

# CC -V
CC: Sun C++ 5.9 SunOS_i386 Patch 124864-09 2008/12/16

# CC -o mysqlconnectorc++client -g0 -xO4 -features=extensions -
I/opt/coolstack/mysql_32bit/include/mysql \
         -I/export/expts/MySQLConnectorC++/include/cppconn -L/opt/coolstack/mysql_32bit/lib/mysql \
         -L/export/expts/MySQLConnectorC++/lib -lmysqlclient_r -lmysqlcppconn
MySQLConnectorC++Client.cpp
```

```
# export
LD_LIBRARY_PATH=/opt/coolstack/mysql_32bit/lib/mysql:/export/expts/ConnectorC++/lib/:$LD_LIBRARY_PATH

# ./mysqlconnectorc++client localhost root admin test

Database connection's autocommit mode = 0

Database Metadata
-----------------
Database Product Name: MySQL
Database Product Version: 5.1.24-rc-standard
Database User Name: root@localhost

Driver name: MySQL Connector/C++
Driver version: 1.0.5

Database in Read-Only Mode?: false
Supports Transactions?: true
Supports DML Transactions only?: false
Supports Batch Updates?: true
Supports Outer Joins?: true
Supports Multiple Transactions?: true
Supports Named Parameters?: false
Supports Statement Pooling?: false
Supports Stored Procedures?: true
Supports Union?: true

Maximum Connections: 151
Maximum Columns per Table: 512
Maximum Columns per Index: 16
Maximum Row Size per Table: 2147483639 bytes

Database schemas:

Total number of schemas = 4

        1. information_schema
        2. ISVe
        3. mysql
        4. test

Executing the Query: "SELECT * FROM City" ..
Retrieving the result set ..

Retrieved 3 row(s).

CityName
--------
Hyderabad, India
San Francisco, USA
Sydney, Australia

ResultSet Metadata
------------------

Number of columns in the result set = 1

    Column Name/Label          Column Type          Column Size
             CityName               VARCHAR                   30


Column "CityName" belongs to the Table: "City" which belongs to the Schema: "test"

Demonstrating Prepared Statements ..
```

```
        Inserting "London, UK" into the table, City ..
        Creating a save point "SAVEPT1" ..
        Inserting "Paris, France" into the table, City ..
        Rolling back until the last save point "SAVEPT1" ..
        Committing outstanding updates to the database ..

Querying the City table again ..

Retrieved 4 row(s).

CityName
--------
Hyderabad, India
San Francisco, USA
Sydney, Australia
London, UK

Cleaning up the resources ..
```

Some of the important steps in the above code sample are explained below.

Establish a Connection to the MySQL Server

A connection to the MySQL Server is established by retrieving an instance of `sql::Connection` from the `sql::Driver` object. An `sql::Driver` object is returned by the `sql::Driver::get_driver_instance` method, and `sql::Driver::connect` method returns the `sql::Connection` object.

Note:

`<namespace>::<classname>::<methodname>` notation was used in the above paragraph to show the fully qualified method name. For example, `insql::Driver::get_driver_instance()`, sql is the namespace, `Driver` is the class name and `get_driver_instance()` is the name of the method. In C++ applications that use `Connector/C++`, you could include "`using namespace sql;`" at the top of your code to avoid having to prefix all Connector/C++ declarations with "`sql::`". For the remainder of this tutorial, the namespace, sql, was omitted from all Connector/C++ specific declarations in favor of simplicity.

Signatures for `Driver::get_driver_instance` and `Driver::connect` methods are shown below. Check the driver.h header in your Connector/C++ installation for the complete list of methods.

```
/* driver.h */
Driver* Driver::get_driver_instance()

Connection* Driver::connect(const std::string& URL, const std::string& userName, const
std::string& password)
Connection* Driver::connect(std::map<std::string, ConnectPropertyVal> properties)
```

The `connect` method was overloaded in the `Driver` class. There are two forms of the `connect` method. In one form, `connect` accepts a database connection URL along with the database username and the password. The second form accepts a `std::map` that contains the connection URL, database username and the password as the key/value pairs.

You can connect to the MySQL Server using TCP/IP by specifying "`tcp://[hostname[:port]][/schemaname]`" for the connection URL. eg., `tcp://127.0.0.1:5555/some_schema`. Both hostname and the port number are optional, and defaults to 127.0.0.1 and 3306 respectively. During runtime, `localhost` will automatically be translated to `127.0.0.1`. Specifying the schema name in connection URL is optional as well, and if not set, be sure to select the database schema using`Connection::setSchema` method.

If you want to use UNIX domain socket to connect to the MySQL Server running on the localhost, specify "`unix://path/to/unix_socket_file`" for the database connection URL. eg., `unix:///tmp/mysql.sock`.

On Windows, you can use named pipes to connect to the MySQL Server running on the localhost by specifying the string "`pipe://path/to/the/pipe`" for the database connection URL. To enable the support for named pipes, you must start the MySQL Server with the `--enable-named-pipe` option. If you don't specify a name to the pipe using the server `option--socket=name`, a named pipe with the default name `MySQL` is created. The name of the pipe is case insensitive on Microsoft Windows.

The following code snippet attempts to connect to the MySQL Server running on the localhost on the default port 3306, using the database username `root`, password `admin` and schema name `test`.

```
using namespace sql;

Driver *driver;
Connection *con;

try {

    driver = get_driver_instance();
    con = driver -> connect("tcp://127.0.0.1:3306/test", "root", "admin");

} catch (..) {
    ..
}
```

`connect` can also be called as shown below using the second form of the overloaded method. `ConnectPropertyVal` is of type `union`, defined in connection.h header. Include the header <map> when compiling the example with this alternate code.

```
..
std::map conn_properties;
ConnectPropertyVal tmp;

tmp.str.val = "unix:///tmp/mysql.sock";
conn_properties [std::string("hostName")] = tmp;

tmp.str.val = "root";
conn_properties [std::string("userName")] = tmp;

tmp.str.val = "admin";
conn_properties [std::string("password")] = tmp;

try {

    driver = get_driver_instance();
    con = driver -> connect(conn_properties);

} catch(..) {
    ..
}
```

If you prefer to separate the protocol from the path to the UNIX socket in the above code, rewrite the line that has the database connection URL as shown below.

```
tmp.str.val = "unix://" "/tmp/mysql.sock";
```

Once the connection has been established, you can use `Connection::setSessionVariable` method for setting variables like `sql_mode`.

C++ Specific Note

The C++ statement `sql::Connection *con = driver -> connect("tcp://127.0.0.1:3306", "root", "admin");` can be re-written using `auto_ptr` template class as shown below.

```
std::auto_ptr < sql::Connection > con ( driver -> connect("tcp://127.0.0.1:3306", "root",
"admin") );
```

= OR =

```
use namespace std;
use namespace sql;

auto_ptr < Connection > con ( driver -> connect("tcp://127.0.0.1:3306", "root", "admin") );
```

The C++ standard library class template `auto_ptr` helps developers manage dynamic memory, and prevent memory leaks in case of unexpected events such as exceptions which would cause the normal cleanup code to be skipped. An `auto_ptr` object has the same semantics as that of a pointer - however when it goes out of scope, it automatically releases the dynamic memory it manages. That is, when `auto_ptr` template class is in use, you need not free up the memory explicitly using the `delete` operator. eg., `delete con;`

To take advantage of `auto_ptr`, you will need to include the <memory> header. This will give access to the std namespace, in which the template class `auto_ptr<type>` resides. The *type* is the data/object type you want your pointer to point to.

The choice of adopting the `auto_ptr` smart pointer mechanism or the traditional mechanism for the dynamic memory management is left to reader's discretion.

Obtain a Statement Object

When the `Connection::createStatement` method is called, it returns a `Statement` object that could be used to send SQL statements to the database server. SQL statements without parameters are normally executed using `Statement` objects. In other words, a `Statement` object is used to execute a static SQL statement and to return the results that it produces. If the same SQL statement has to be executed multiple times with different inputs, consider using Prepared Statements for efficiency during the runtime.

Signature for the `Connection::createStatement` method is shown below. For the complete list of methods supported by the `Connection` interface, check the connection.h header in your Connector/C++ installation.

```
/* connection.h */
Statement* Connection::createStatement();
```

The following code fragment invokes the `createStatement` method of the `Connection` interface to obtain an object of type `Statement`.

```
Connection *con;
Statement *stmt;

Statement stmt = con -> createStatement();
```

In this example, `con` is a reference to an object of type `Connection`.

Execute the SQL Statements

Before you can execute the SQL statements over the database connection, you have to select appropriate database schema. To select the database schema, call the setSchema method of the Connection object with the schema name as the argument.

To execute the `SELECT` statements, call the `Statement::executeQuery` method with the SQL statement as the argument. `executeQuery()` returns a `ResultSet` object. The`Statement::executeUpdate` method can be used to execute the given SQL statement, which may be an `INSERT, UPDATE, DELETE`, or any `SQL` SQL statement that returns nothing, such as an SQL DDL statement. Unlike `executeQuery()`, the `executeUpdate` method does not return a ResultSet. Instead, it returns the number of rows affected by the `INSERT, UPDATE`, or `DELETE` statements.

If you don't know ahead of time whether the SQL statement will be a `SELECT` or an `INSERT, UPDATE`,or `DELETE`, use the `execute` method. `execute()` returns true if the SQL query was a `SELECT`, and returns `false` if the statement was an `INSERT, UPDATE`,or `DELETE`. If the statement was a `SELECT` query, you can retrieve the results by calling the getResultSetmethod on the Statement instance. If the statement was an `INSERT, UPDATE`,or `DELETE` statement, you can retrieve the count of affected rows by calling `getUpdateCount()`.

In some uncommon situations, a single SQL statement may return multiple result sets and/or update counts. Normally you can ignore this unless you are executing a stored procedure that you know may return multiple results or you are dynamically executing an unknown SQL statement. In such cases, use the methods `getResultSet` or `getUpdateCount` to retrieve the result, and `getMoreResults()` to determine whether there is another result set.

Signatures for some of the relevant methods are shown below. For the complete list of methods supported by `Connection` and `Statement` interfaces, check the connection.h and statement.h headers in your Connector/C++ installation.

```
/* connection.h */
void Connection::setSchema(const std::string& catalog);

/* statement.h */
ResultSet* Statement::executeQuery (const std::string& sql);
int Statement::executeUpdate (const std::string& sql);
bool Statement::execute (const std::string& sql);

ResultSet* Statement::getResultSet();
uint64_t Statement::getUpdateCount();
```

All the above methods throw the `SQLException`, so make sure to catch those exceptions in your code. Just for the sake of simplicity, the code fragments in the examples are not surrounded by the `try .. catch`blocks.

If you look again at the full sample code, you will see that the objective is to retrieve all the rows from the `City` table in the `test` database. Hence `executeQuery()` was used in the sample code as shown here:

```
Statement *stmt;
ResultSet *res;

res = stmt -> executeQuery ("SELECT * FROM City");
```

The method `executeQuery` returns a `ResultSet` object that contains the data generated by the given query. This method throws `SQLException` in case of a database access error or if this method is called on a closed `Statement` or the given SQL statement produces anything other than a single `ResultSe` object.

Alternatively, the previous code fragment can be re-written using `Statement::execute()` as shown below.

```
bool retvalue = stmt -> execute ("SELECT * FROM City");

if (retvalue) {
    res = stmt -> getResultSet();
} else {
    ...
}
```

The method `execute` returns `true` if the first result is a `ResultSet` object, and `false` if it is an update count or there are no results. If `execute()` returns `true`, retrieve the result set using the `getResultSet` method. `getResultSet()` returns `NULL` if the result is an update count or if there are no more results. `getResultSet` method should be called only once per result.

Both the methods, `execute` and `getResultSet`, throw the SQLException in case of a database access error or if this method is called on a closed Statement.

If you want to insert new records into the database, you can do so with the help of `executeUpdate` method as shown in the following example:

```
int updateCount = stmt -> executeUpdate ("INSERT INTO City (CityName) VALUES
('Napier, New Zealand')");
```

The method `executeUpdate` executes the given SQL Data Manipulation Language (DML) statement, such as `INSERT, UPDATE` or `DELETE`, or an SQL statement that returns nothing, such as a DDL statement. `executeUpdate()` returns the row count for `INSERT, UPDATE` or `DELETE` statements and 0 for SQL statements that return nothing.

`executeUpdate()` throws SQLException in case of a database access error or when it is called on a closed Statement or the given SQL statement produces a ResultSet object.

Another way to re-write the above code fragment using `execute` and `getUpdateCount` methods:

```
int updateCount = 0;

bool retstatus = stmt -> execute ("INSERT INTO City (CityName) VALUES
('Napier, New Zealand')");

if (!retstatus) {
    updateCount = stmt -> getUpdateCount();
} else {
    ...
}
```

The method `execute` returns `false` if the first result is an update count or if there are no results. If the first result is an update count , retrieve the value using the `getUpdateCount` method. `getUpdateCount()` returns -1 if the current result is a `ResultSet` object or if there are no more results. `getUpdateCount()` should be called only once per result.

Both the methods, `execute` and `getUpdateCount`, throw the `SQLException` in case of a database access error or if called on a closed `Statement`.

Retrieve the Data From the Result Set

The previous paragraphs explained that the methods for executing the SQL queries, `executeQuery` and `execute`, return an instance of `ResultSet`. You can use the `ResultSet` object to access the data that was returned by running a query against the database. Every `ResultSet` object maintains a cursor, which points to its current row of data. The rows in the result set cursor are retrieved in sequence. Within a row, column values can be accessed in any order. You can refer to the columns by their position (offset) or by their name/label, though the latter is fault-tolerant to some extent, especially when there are changes to the table schema. Reference by the column label or name makes the code lucid. On the other hand, using the column index, or positional reference, may improve the performance.

Column label is the label for the column specified with the SQL "AS" clause. If the SQL "AS" clause is not specified, then the label is the name of the column. For example, CN is the column label in the SQL statement: `SELECT CityName AS CN FROM City`.

The data stored in the `ResultSet` can be retrieved by using various `getXX` methods such as `getString()` or `getInt()`, depending on the type of data being retrieved. Use the next andprevious methods of the `ResultSet` object to move the cursor to the next and previous rows in the result set.

The `ResultSet` object remains open even when the `Statement` object that generated it is closed, re-executed, or used to retrieve the next result from a sequence of multiple results. Once the result set had been pulled out of the `Statement`, the `ResultSet` object remains valid until it is closed explicitly or implicitly, irrespective of the state of the `Statement` object that generated it.

As of this writing, MySQL Connector/C++ returns buffered results for the `Statement` objects. Buffered result sets are cached on the client. The driver will always fetch all the data no matter how big the result set is. Future versions of the connector is expected to return buffered and unbuffered results for the `Statement` objects.

Signatures for some of the relevant methods are shown below. For the complete list of methods supported by the `ResultSet` interface, check the resultset.h headers in your Connector/C++ installation.

```
/* resultset.h */
size_t ResultSet::rowsCount() const;
void ResultSet::close();

bool ResultSet::next();
bool ResultSet::previous();
bool ResultSet::last();
bool ResultSet::first();

void ResultSet::afterLast();
void ResultSet::beforeFirst();

bool ResultSet::isAfterLast() const;
bool ResultSet::isBeforeFirst()const;
bool ResultSet::isClosed() const;

bool ResultSet::isNull(uint32_t columnIndex) const;
bool ResultSet::isNull(const std::string& columnLabel) const;
bool ResultSet::wasNull() const;

std::string ResultSet::getString(uint32_t columnIndex) const;
std::string ResultSet::getString(const std::string& columnLabel) const;

int32_t ResultSet::getInt(uint32_t columnIndex) const;
int32_t ResultSet::getInt(const std::string& columnLabel) const;
```

In the sample C++ code, the query "`SELECT * FROM City`" returns a `ResultSet` with only one column `CityName` of data type String, known as `VARCHAR` in MySQL.

The following code fragment loops through the `ResultSet` object res, retrieves the CityName from each row by referencing the exact column name, and displays it on the standard output:

```
while (res -> next()) {
    cout << rs -> getString("CityName") << endl;
}
```

Because it is also possible to refer to the column by its position, the following code fragment produces similar results:

```
while (res -> next()) {
         cout << rs -> getString(1) << endl;
}
```

The integer argument to `getString()` refers to the position of the column in the list of columns specified in the query, starting with 1 for the first field.

Both versions of `getString()` return the column value. If the column value is SQL NULL, the value returned is an empty string. You can use `ResultSet::isNull` method with the column offset or column name/label as the argument to check whether the column value being fetched is a SQL NULL. To determine whether the last column read had a value of SQL NULL, call `ResultSet::wasNull` with no arguments.

The following example shows how to fetch the data by traversing the cursor in reverse order.

```
/* Move the cursor to the end of the ResultSet object, just after the last row */
res -> afterLast();

if (!res -> isAfterLast()) {
    throw runtime_error("Error: Cursor position should be at the end of the result set
    after the last row.");
}

/* fetch the data : retrieve all the rows in the result set */
while (res -> previous()) {
    cout << rs -> getString("CityName") << endl;
}
```

The method `getString` throws `SQLException` if the column label/name is not valid or if the column index is not valid, in case of a database access error or if this method is called on a closed result set.

Using Prepared Statements

MySQL 4.1 introduced prepared statements to accomplish the task of executing a query repeatedly, albeit with different parameters in each iteration. Prepared Statements can help increase security by separating SQL logic from the data being supplied. This separation of logic and data can help prevent a very common type of vulnerability known as SQL injection attack. However note that although prepared statements can improve the security, it is still the responsibility of the application developer to guard against security attacks and to sanitize the input before submitting to the database for processing.

Prepared Statements are optimized for handling parameterized SQL statements that can benefit from pre-compilation. Unlike `Statement` object, an SQL statement is provided to the `PreparedStatement` object when it is created. In most cases, the SQL statement is sent to the database server right away, where the query parser performs the syntax check, syntactic optimization and finally parses (compiles) the SQL for later use by the MySQL Server . As a result, the `PreparedStatement` object contains an SQL statement that has been

precompiled. This means when the prepared statement is executed, the database server can just run the `PreparedStatement` SQL statement without having to compile it first. Reduced query parsing may lead to significant performance improvements in the MySQL Server.

The MySQL client/server protocol supports two methods of sending the database results to the client: as text and as binary. The text protocol always converts the data into strings before sending them across the network, and the server decodes the strings into appropriate data types. Unlike the text protocol, the binary protocol avoids converting the data into strings wherever possible. The binary protocol is used only with the prepared statements. Based on the data being sent over the network, the binary protocol that the prepared statements use can reduce the CPU and network overhead by eliminating the encoding and decoding of the strings into correct data types at the client and the server.

Connector/C++ is based on the MySQL C API and the C library `libmysql`. Therefore it inherits all limitations from the MySQL Server and the MySQL C API. The following statements can be used as prepared statements with Connector/C++: `CALL`, `CREATE TABLE`, `DELETE`, `DO`, `INSERT`, `REPLACE`, `SELECT`, `SET`, `UPDATE`, and majority of the SHOWstatements. USE is not supported by the prepared statement protocol, and Connector/C++ does not include a prepared statement emulation for the USE statement. Check the MySQL C API Prepared Statements documentation for the complete list of statements which can be prepared.

Although `PreparedStatement` objects can be used with SQL statements with no parameters, you probably use them often with parameterized SQL statements.

Create a PreparedStatement Object

Just like `Statement` objects, you can create `PreparedStatement` objects by using the `Connection` instance. Calling the prepare `Statement` method against an active database `Connection` object creates a `PreparedStatement` object for sending parameterized SQL statements to the database.

Method signature:

```
/* connection.h */
PreparedStatement * Connection::prepareStatement(const std::string& sql) throws
SQLException;
```

`preparedStatement()` returns a new default `PreparedStatement` object containing the pre-compiled SQL statement. This method throws `SQLException` in case of a database access error or when this method is called on a closed connection.

The following code fragment in the sample C++ code creates the `PreparedStatement` object.

```
Connection *con;
PreparedStatement *prep_stmt;

..
prep_stmt = con -> prepareStatement ("INSERT INTO City (CityName) VALUES (?)");
```

Supply Values for PreparedStatement Parameters

In case of parameterized SQL statements, you need to supply the values to be used in place of the question mark placeholders before you can execute the SQL statement. You can do this by calling one of the `setXX` methods defined in the `PreparedStatement` class. The setter methods (`setXX()`) for setting IN parameter values must specify types that are compatible with the defined SQL type of the input parameter. For instance, if the IN parameter has SQL type INTEGER, then `setInt()` should be used.

The following code fragment in the sample C++ code sets the question mark placeholder to a C++ `std::string` with a value of "`London, UK`".

```
PreparedStatement *prep_stmt;

..
prep_stmt -> setString (1, "London, UK");
```

Execute the SQL statement in the PreparedStatement Object

Similar to the execution of SQL statements with the `Statement` object, you can use `executeQuery` method to execute `SELECT` statements, `executeUpdate` method to execute SQL statements which may be an `INSERT`, `UPDATE`, `DELETE` or an SQL statement that returns nothing such as an SQL DDL statement, and execute method to execute SQL statement of any kind. Check the Execute the SQL Statements paragraph for more details.

Signatures for some of the relevant methods are shown below. For the complete list of methods supported by `PreparedStatement` interface, check the prepared_statement.h header in your Connector/C++ installation.

```
/* prepared_statement.h */
ResultSet* PreparedStatement::executeQuery();
int PreparedStatement::executeUpdate();
bool PreparedStatement::execute();

ResultSet* PreparedStatement::getResultSet();
uint64_t PreparedStatement::getUpdateCount();
```

All the above methods throw the `SQLException`, so make sure to catch those exceptions in your code.

The `ResultSet` object remains open even when the `PreparedStatement` object that generated it is re-executed, or used to retrieve the next result from a sequence of multiple results. Once the result set had been pulled out of the `PreparedStatement`, the `ResultSet` object remains valid until it is closed explicitly or implicitly, or the `PreparedStatement` object that generated it is closed, whichever occurs first.

The following code fragment in the sample C++ code executes the SQL statement in the `PreparedStatement` object.

```
..
prep_stmt -> setString (1, "London, UK");
int updatecount = prep_stmt -> executeUpdate();
```

In general, parameter values remain in force for repeated use of a statement. Setting a parameter value automatically clears its previous value. However, in some cases it is useful to release all the resources used by the current parameter values at once. This can be done by calling the method, `PreparedStatement::clearParameters`.

Using Transactions

A database transaction is a set of one or more statements that are executed together as a unit, so either all of the statements are executed, or none of them are executed. MySQL supports local transactions within a given client session through statements such as SET autocommit, `START TRANSACTION`, `COMMIT`, and `ROLLBACK`.

Disable AutoCommit Mode

By default, all the new database connections are in autocommit mode. In the autocommit mode, all the SQL statements will be executed and committed as individual transactions. The commit occurs when the statement completes or the next execute occurs, whichever comes first. In case of statements that return a `ResultSet` object, the statement completes when the last row of the `ResultSet` object has been retrieved or the `ResultSet` object has been closed.

One way to allow multiple statements to be grouped into a transaction is to disable autocommit mode. In other words, to use transactions, the `Connection` object must not be in autocommit mode. The `Connection` class provides the `setAutoCommit` method to enable or disable the autocommit. An argument of 0 to `setAutoCommit()` disables the autocommit, and a value of 1 enables the autocommit.

```
Connection *con;
..
/* disable the autocommit */
con -> setAutoCommit(0);
```

It is suggested to disable autocommit only while you want to be in transaction mode. This way, you avoid holding database locks for multiple statements, which increases the likelihood of conflicts with other users.

Commit or Rollback a Transaction

Once autocommit is disabled, changes to transaction-safe tables such as those for InnoDB and NDBCLUSTER are not made permanent immediately. You must explicitly call the method `commit` to make the changes permanent in the database or the method `rollback` to undo the changes. All the SQL statements executed after the previous call to `commit()` are included in the current transaction and committed together or rolled back as a unit.

The following code fragment, in which `con` is an active connection, illustrates a transaction.

```
Connection *con;
PreparedStatement *prep_stmt;

..
con -> setAutoCommit(0);

prep_stmt = con -> prepareStatement ("INSERT INTO City (CityName) VALUES (?)");

prep_stmt -> setString (1, "London, UK");
prep_stmt -> executeUpdate();

con -> rollback();

prep_stmt -> setString (1, "Paris, France");
prep_stmt -> executeUpdate();

con -> commit();
```

In this example, autocommit mode is disabled for the connection `con`, which means that the prepared statement prep_stmt is committed only when the method `commit` is called against this active connection object. In this case, an attempt has been made to insert two rows into the database using the prepared statement, but the first row with data "`London, UK`" was discarded by calling the `rollback` method while the second row with data "`Paris, France`" was inserted into the `City` table by calling the `commit` method.

Another example to show the alternate syntax to disable the autocommit, then to commit and/or rollback transactions explicitly.

```
Connection *con;
Statement *stmt;

..
stmt = con -> createStatement();

//stmt -> execute ("BEGIN;");
//stmt -> execute ("BEGIN WORK;");
stmt -> execute ("START TRANSACTION;");

stmt -> executeUpdate ("INSERT INTO City (CityName) VALUES ('London, UK')");
stmt -> execute ("ROLLBACK;");

stmt -> executeUpdate ("INSERT INTO City (CityName) VALUES ('Paris, France')");
stmt -> execute ("COMMIT;");
```

The `START TRANSACTION` or `BEGIN` statement starts a new transaction. `COMMIT` commits the current transaction to the database by making the changes permanent. `ROLLBACK` rolls back the current transaction by canceling the changes to the database. With `START TRANSACTION`, autocommit remains disabled until you end the transaction with `COMMIT` or `ROLLBACK`. The autocommit mode then reverts to its previous state.

`BEGIN` and `BEGIN WORK` are supported as aliases of `START TRANSACTION` for initiating a transaction. `START TRANSACTION` is standard SQL syntax and it is the recommended way to start an ad-hoc transaction.

Rollback to a Savepoint within a Transaction

The MySQL connector for C++ supports setting savepoints with the help of `Savepoint` class, which offer finer control within transactions. The `Savepoint` class allows you to partition a transaction into logical breakpoints, providing control over how much of the transaction gets rolled back.

As of this writing, InnoDB and Falcon storage engines support the savepoint transactions in MySQL 6.0.

To use transaction savepoints, the `Connection` object must not be in autocommit mode. When the autocommit is disabled, applications can set a savepoint within a transaction and then roll back all the work done after the savepoint. Note that enabling autocommit invalidates all the existing savepoints, and the Connector/C++ driver throws an `InvalidArgumentException` when an attempt has been made to roll back the outstanding transaction until the last savepoint.

A savepoint is either named or unnamed. You can specify a name to the savepoint by supplying a string to the `Savepoint::setSavepoint` method. If you do not specify a name, the savepoint is assigned an integer ID. You can retrieve the savepoint name using `Savepoint::getSavepointName()`.

Signatures of some of the relevant methods are shown below. For the complete list of methods supported by `Connection, Statement, PreparedStatement` and `Savepoint` interfaces, check the connection.h, statement.h and prepared_statement.h headers in your Connector/C++ installation.

```
/* connection.h */
Savepoint* Connection::setSavepoint(const std::string& name);
void Connection::releaseSavepoint(Savepoint * savepoint);
void Connection::rollback(Savepoint * savepoint);
```

The following code fragment inserts a row into the table `City`, creates a savepoint `SAVEPT1`, then inserts a second row. When the transaction is later rolled back to `SAVEPT1`, the second insertion is undone, but the first insertion remains intact. In other words, when the transaction is committed, only the row containing "`London, UK`" will be added to the table `City`.

```
Connection *con;
PreparedStatement *prep_stmt;
Savepoint *savept;

..
prep_stmt = con -> prepareStatement ("INSERT INTO City (CityName) VALUES (?)");

prep_stmt -> setString (1, "London, UK");
prep_stmt -> executeUpdate();

savept = con -> setSavepoint ("SAVEPT1");

prep_stmt -> setString (1, "Paris, France");
prep_stmt -> executeUpdate();

con -> rollback (savept);
con -> releaseSavepoint (savept);

con -> commit();
```

The method `Connection::releaseSavepoint` takes a `Savepoint` object as a parameter and removes it from the current transaction. Any savepoints that have been created in a transaction are automatically released and become invalid when the transaction is committed, or when the entire transaction is rolled back. Rolling back a transaction to a savepoint automatically releases and invalids any other savepoints that were created after the savepoint in question. Once a savepoint has been released, any attempt to reference it in a rollback operation causes the `SQLException` to be thrown.

Accessing Result Set Metadata

When the SQL statement being processed is unknown until runtime, the `ResultSetMetaData` interface can be used to determine the methods to be used to retrieve the data from the result set. `ResultSetMetaData` provides information about the structure of a given result set. Data provided by the `ResultSetMetaData` object includes the number of columns in the result set, the names or labels and the types of those columns along with the attributes of each column and the names of the table, schema and the catalog the designated column's table belongs to.

When `getMetaData()` is called on a `ResultSet` object, it returns a `ResultSetMetaData` object describing the columns of that `ResultSet` object.

Signatures of some of the relevant methods are shown below. For the complete list of methods supported by `ResultSetMetaData` interface, check the resultset_metadata.h header in your Connector/C++ installation.

```
/* resultset.h */
ResultSetMetaData * ResultSet::getMetaData() const;

/* prepared_statement.h */
ResultSetMetaData * PreparedStatement::getMetaData() const;

/* resultset_metadata.h */
std::string ResultSetMetaData::getCatalogName(unsigned int columnIndex);
std::string ResultSetMetaData::getSchemaName(unsigned int columnIndex);
std::string ResultSetMetaData::getTableName(unsigned int columnIndex);
```

```
unsigned int ResultSetMetaData::getColumnCount();
unsigned int ResultSetMetaData::getColumnDisplaySize(unsigned int columnIndex);
std::string ResultSetMetaData::getColumnLabel(unsigned int columnIndex);
std::string ResultSetMetaData::getColumnName(unsigned int columnIndex);
int ResultSetMetaData::getColumnType(unsigned int columnIndex);
std::string ResultSetMetaData::getColumnTypeName(unsigned int columnIndex);

int ResultSetMetaData::isNullable(unsigned int columnIndex);
bool ResultSetMetaData::isReadOnly(unsigned int columnIndex);
bool ResultSetMetaData::isWritable(unsigned int columnIndex);
```

The following code fragment demonstrates how to retrieve all the column names or labels, their data types and the sizes along with the table name and the schema names to which they belong.

```
ResultSet *rs;
ResultSetMetaData *res_meta;

res_meta = rs -> getMetaData();

int numcols = res_meta -> getColumnCount();
cout << "\nNumber of columns in the result set = " << numcols << endl;

cout.width(20);
cout << "Column Name/Label";

cout.width(20);
cout << "Column Type";

cout.width(20);
cout << "Column Size" << endl;

for (int i = 0; i < numcols; ++i) {
    cout.width(20);
    cout << res_meta -> getColumnLabel (i+1);

    cout.width(20);
    cout << res_meta -> getColumnTypeName (i+1);

    cout.width(20);
    cout << res_meta -> getColumnDisplaySize (i+1) << endl;
}

cout << "\nColumn \"" << res_meta -> getColumnLabel(1);
cout << "\" belongs to the Table: \"" << res_meta -> getTableName(1);
cout << "\" which belongs to the Schema: \"" << res_meta -> getSchemaName(1) << "\"" <<
endl;

//delete res_meta;
delete rs;
```

From release 1.0.5 onwards, the connector takes care of cleaning the `ResultSetMetaData` objects automatically when they go out of scope. This will relieve the clients from deleting the `ResultSetMetaData` objects explicitly. Due to the implicit destruction of the metadata objects, clients won't be able to delete the `ResultSetMetaData` objects directly. Any attempt to delete the `ResultSetMetaData` object results in compile time error. Similarly using `auto_ptr` template class to instantiate an object of type `ResultSetMetaData` results in compile time error. For example, compilation of the above code fails with Connector/C++ 1.0.5 and later versions, when the statement `delete res_meta;` is uncommented.

Prepared Statements and the Result Set Metadata

`PreparedStatement::getMetaData()` retrieves a `ResultSetMetaData` object that contains information about the columns of the `ResultSet` object, which will be returned when the `PreparedStatement` object is executed.

Because a `PreparedStatement` object is precompiled, it is possible to know about the `ResultSet` object that it will return without having to execute it. Consequently, it is possible to invoke the method `getMetaData` on a `PreparedStatement` object rather than waiting to execute it and then invoking the `ResultSet::getMetaData` method on the `ResultSet` object that is returned.

The method `PreparedStatement::getMetaData` is supported only in Connector/C++ 1.0.4 and later versions.

Accessing Database Metadata

You can retrieve general information about the structure of a database with the help of `DatabaseMetaData` interface. `DatabaseMetaData` exposes a significant amount of information about the support and contents of a given database. For instance, by using the `DatabaseMetaData` interface, it is possible to find out whether the database supports transactions, outer joins are supported and to what extent, the maximum number of allowed concurrent connections to remain open, and the `ResultSet` types supported.

A user for this interface is commonly a tool that needs to discover how to deal with the underlying Database Management System (DBMS). This is especially true for applications that are intended to be used with more than one DBMS. For example, a tool might use `getTypeInfo()` to find out what data types can be used in a `CREATE TABLE` statement. A user may call `supportsCorrelatedSubqueries()` and `supportsBatchUpdates()` to see if it is possible to use a correlated subquery and if batch updates are allowed.

Database metadata is associated with a particular connection, so objects of the `DatabaseMetaData` class are created by calling the `getMetaData` method on an active `Connection` object.

Signatures of some of the methods in `DatabaseMetaData` interface are shown below. For the complete list of methods supported by `DatabaseMetaData` interface, check the metadata.h header in your Connector/C++ installation.

```
 &types);

ResultSet *DatabaseMetaData::getIndexInfo(const std::string& catalog, const std::string&
schema, const std::string& table, bool unique, bool approximate);

int DatabaseMetaData::getMaxColumnsInIndex();
int DatabaseMetaData::getMaxColumnsInTable();
int DatabaseMetaData::getMaxConnections();

ResultSet *DatabaseMetaData::getSchemas();
ResultSet *DatabaseMetaData::getTableTypes();

bool DatabaseMetaData::supportsTransactions();
bool DatabaseMetaData::supportsBatchUpdates();
bool DatabaseMetaData::supportsMultipleResultSets();
bool DatabaseMetaData::supportsNamedParameters();
bool DatabaseMetaData::supportsSavepoints();
bool DatabaseMetaData::supportsStatementPooling();
bool DatabaseMetaData::supportsStoredProcedures();
bool DatabaseMetaData::supportsUnion();

/* connection.h */
DatabaseMetaData *Connection::getMetaData();
```

```
/* metadata.h */
const std::string& DatabaseMetaData::getDatabaseProductName();
std::string DatabaseMetaData::getDatabaseProductVersion();

const std::string& DatabaseMetaData::getDriverName();
const std::string& DatabaseMetaData::getDriverVersion();

ResultSet *DatabaseMetaData::getTables(const std::string& catalog, const std::string&
schemaPattern, const std::string& tableNamePattern, std::list
```

The following code fragment demonstrates how to retrieve some of the metadata from the database using the methods outlined above.

```
Connection *dbcon;

/* the following line results in compilation error with Connector/C++ 1.0.5 and later */
//auto_ptr < DatabaseMetaData > dbcon_meta ( dbcon -> getMetaData() );

DatabaseMetaData *dbcon_meta = dbcon -> getMetaData();

cout << boolalpha;

cout << "Database Product Name: " << dbcon_meta -> getDatabaseProductName() << endl;
cout << "Database Product Version: " << dbcon_meta -> getDatabaseProductVersion() << endl;
cout << "Driver name: " << dbcon_meta -> getDriverName() << endl;
cout << "Driver version: " << dbcon_meta -> getDriverVersion() << endl << endl;
cout << "Supports Transactions?: " << dbcon_meta -> supportsTransactions() << endl;
cout << "Supports Named Parameters?: " << dbcon_meta -> supportsNamedParameters() << endl;
cout << "Maximum Connections: " << dbcon_meta -> getMaxConnections() << endl;
cout << "Maximum Columns per Table: " << dbcon_meta -> getMaxColumnsInTable() << endl;

cout << "\nDatabase schemas: " << endl;

auto_ptr < ResultSet > rs ( dbcon_meta -> getSchemas() );

cout << "\nTotal number of schemas = " << rs -> rowsCount() << endl;

int row = 1;

while (rs -> next()) {
    cout << "\t" << row << ". " << rs -> getString("TABLE_SCHEM") << endl;
    ++row;
} // while
```

From release 1.0.5 onwards, the connector takes care of cleaning the `DatabaseMetaData` objects automatically when they go out of scope. This will relieve the clients from deleting the `DatabaseMetaData` objects explicitly. Due to the implicit destruction of the metadata objects, clients won't be able to delete the `DatabaseMetaData` objects directly. Any attempt to delete the `DatabaseMetaData` object results in compile time error. Similarly using `auto_ptr` template class to instantiate an object of type `DatabaseMetaData` results in compile time error.

Accessing Parameter Metadata from a PreparedStatement Object

Connector/C++ has partial support for returning parameter metadata for a given `PreparedStatement` object. As of this writing, Connector/C++ has support for only one method, `getParameterCount`, in the `ParameterMetaData` interface. Future versions of Connector/C++ may support majority of the methods defined in the `ParameterMetaData` interface in JDBC 4.0 specification.

The method `getParameterCount` returns the number of parameters in the `PreparedStatement` object for which the `ParameterMetaData` object would contain information. `getParameterCount()` throws `SQLException` in case of a database access error.

Signatures of the relevant methods are shown below.

```
/* prepared_statement.h */
ParameterMetaData* PreparedStatement::getParameterMetaData();

/* parameter_metadata.h */
int ParameterMetaData::getParameterCount();
```

The method `getParameterCount` was declared in parameter_metadata.h header in your Connector/C++ installation. Make sure to include parameter_metadata.h header file in your C++ application, if your code has references to the `ParameterMetaData::getParameterCount` method.

The following code fragment demonstrates how to retrieve the parameter count from a given `PreparedStatement` object.

```
#include <cppconn/driver.h>
#include <cppconn/prepared_statement.h>
#include <cppconn/parameter_metadata.h>

...
...

Driver *driver;
Connection    *con;
PreparedStatement *prep_stmt;
ParameterMetaData *param_meta;

try {

    driver = get_driver_instance();
    con = driver -> connect("tcp://127.0.0.1:3306", "root", "admin");

    prep_stmt = con -> prepareStatement ("INSERT INTO City (CityName) VALUES (?)");

    param_meta = prep_stmt -> getParameterMetaData();
    cout << "Number of parameters in the prepared statement = "
    << param_meta -> getParameterCount() << endl;

    //delete param_meta;

} catch (..) {
    ...
}
```

From release 1.0.5 onwards, the connector takes care of cleaning the `ParameterMetaData` objects automatically when they go out of scope. This will relieve the clients from deleting the `ParameterMetaData` objects explicitly. Due to the implicit destruction of the metadata objects, clients won't be able to delete the `ParameterMetaData` objects directly. Any attempt to delete the `ParameterMetaData` object results in compile time error. Similarly using `auto_ptr` template class to instantiate an object of type `ParameterMetaData` results in compile time error. For example, compilation of the above code fails with Connector/C++ 1.0.5 and later versions, when the statement `delete param_meta;` is uncommented.

Clean Up: Release the System Resources

If an object is no longer needed, destroy that object by using the C++ `delete` operator. It is not necessary to explicitly release the memory resources, if you are using the `auto_ptr` template class to manage the memory resources dynamically. Check C++ Specific Note for more details about the `auto_ptr` smart pointer. In all other cases, ensure you explicitly release the allocated memory even in the case of runtime errors and exceptions. Failure to do so results in memory leaking.

Similarly free up the system resources by closing files, destroying threads, etc., when they are no longer needed in the context of the application.

The following code fragment deletes the `ResultSet, Statement, PreparedStatement` and `Connection` objects `res, stmt, prep_stmt` and `con` respectively.

```
Connection *con;
Statement *stmt;
PreparedStatement *prep_stmt;
ResultSet *res;
ResultSetMetaData *res_meta;
DatabaseMetaData *dbcon_meta;

delete res;
delete stmt;
delete prep_stmt;
//delete res_meta;
//delete dbcon_meta;
con -> close();
delete con;
```

Starting with the release of Connector/C++ 1.0.5, clients are no longer required to destruct the metadata objects directly. The driver takes care of the metadata object destruction for the clients.

When autocommit is disabled in MySQL, all the uncommitted transactions will be rolled back automatically if you close the database connection without calling the commit method explicitly.

Catching Exceptions

In addition to the `std::runtime_error`, the MySQL Connector/C++ can throw four different exceptions that can provide information about a database error or other errors.

1. `SQLException`, which was derived from `std::runtime_error`
2. `InvalidArgumentException`, which was derived from `SQLException`
3. `MethodNotImplementedException`, which was derived from `SQLException`, and
4. `InvalidInstanceException`, which was derived from `SQLException`.

All the above exceptions are defined in exception.h header in your Connector/C++ installation.

Since all those exception classes are derived from `std::runtime_error` directly or indirectly, they can return a C-style character string describing the general cause of the current error when the method `what` is called against an exception object.

The `SQLException` class has implementation for two other methods - 1. `getSQLState()`, which returns the current SQL state in the form of a string; and 2. `getErrorCode()`, which returns an integer error code that corresponds to the MySQL error code. Check MySQL Reference Manual for the Server Error Codes and Messages.

The following outputs demonstrate Connector/C++'s ability to throw exceptions in case of incorrect inputs.

```
# ./mysqlconnectorc++client localhost giri admin test
ERROR: SQLException in MySQLConnectorC++Client.cpp (main) on line 255
ERROR: Access denied for user 'giri'@'localhost' (using password: YES) (MySQL error code:
1045, SQLState: 28000)

# ./mysqlconnectorc++client localhost root admin test2
Database connection's autocommit mode = 0
ERROR: SQLException in MySQLConnectorC++Client.cpp (main) on line 255
ERROR: Unknown database 'test2' (MySQL error code: 1049, SQLState: 42000)

# ./mysqlconnectorc++client ben13.sfbay root admin test
ERROR: SQLException in MySQLConnectorC++Client.cpp (main) on line 255
ERROR: Unknown MySQL server host 'ben13.sfbay' (1) (MySQL error code: 2005, SQLState:
HY000)

When a non-existing column name is specified:

# ./mysqlconnectorc++client localhost root admin test
...
InvalidArgumentException: MySQL_ResultSet::getString: invalid value of 'columnIndex'
...
```

Debug Tracing with MySQL Connector/C++

In case of random problems, debug traces and protocol files generated by the connector driver are more useful for the problem diagnosis than the traditional tools like a debugger to debug your client application.

The MySQL Connector/C++ is capable of generating two types of debug traces.

1. Traces generated by the debug version of MySQL Client Library, and
2. Traces generated internally by the Connector/C++ driver

As of this writing, the debug traces can only be activated through API calls in the case of the debug version of MySQL Client Library. Those traces are controlled on a per-connection basis. On the other hand, connector's internal traces can be activated for the whole client application by setting the environment variable, `MYSQLCPPCONN_TRACE_ENABLED`, on non-Windows platforms. In both of those cases, you can use `Connection::setClientOptions()` to activate and deactivate the debug traces selectively for certain function calls. By default, the MySQL Client Library trace is always written to the standard error (stderr), whereas the Connector/C++ drivers' protocol messages are written to the standard output (stdout).

Be advised that the debug traces may contain SQL statements from your application. Exercise caution when turning the trace functionality on, especially when the SQL statements should not be exposed at the client side.

Signature of the `setClientOptions` method is shown below. This method was defined in connection.h header in your Connector/C++ installation.

```
/* connection.h */
void Connection::setClientOption(const std::string & optionName, const void * optionValue);
```

The string "libmysql_debug" for the option name enables the debug trace in the case of debug version of MySQL Client Library, where as the string "clientTrace" enables the debug trace in the case of internal tracing by the Connector/C++ driver.

Traces Generated by the MySQL Client Library

The connector driver implicitly calls the C-API function, mysql_debug, to enable this debug trace. Only debug version of the MySQL Client Library is capable of generating the debug trace. Therefore to enable the trace functionality, link the Connector/C++ driver against the debug version of the MySQL Client Library. It is not necessary to build the driver with -DMYSQLCPPCONN_TRACE_ENABLE:BOOL=1 CMake option or to set the environment variable, MYSQLCPPCONN_TRACE_ENABLED, in the runtime to generate this trace. The trace shows the internal function calls and the addresses of internal objects as shown below.

```
<cli_read_query_result
>mysql_real_query
| enter: handle: 0x808a228
| query: Query = 'SELECT CityName as CITYNAME FROM City'
| >mysql_send_query
| | enter: rpl_parse: 0  rpl_pivot: 1
| <mysql_send_query
| >cli_advanced_command
| | >net_write_command
| | | enter: length: 37
| | <net_write_command
| | >net_flush
| | | <vio_is_blocking
| | | >net_real_write
| | | | >vio_write
| | | | | enter: sd: 4  buf: 0x808fa38  size: 42
| | | | | exit: 42
...
...
Voluntary context switches 25, Involuntary context switches 5
>TERMINATE
| safe: sf_malloc_count: 4
| safe: Memory that was not free'ed (16420 bytes):
| safe: Maximum memory usage: 65569 bytes (65k)
<TERMINATE
```

The following code fragment demonstrates how to generate debug trace for the executeQuery function call using MySQL Client Library and Connector/C++'s Connection::setClientOptions().

```
Driver *driver;
Connection *con;
Statement *stmt;
ResultSet *rs;

string url, user, password;

try {
    driver = get_driver_instance();

    con = driver -> connect(url, user, password);
    stmt = con -> createStatement();

    con -> setClientOption ("libmysql_debug", "d:t:O,/tmp/client.trace");

    rs = stmt -> executeQuery ("SELECT * FROM City");

    con -> setClientOption ("libmysql_debug", "f");

} catch (SQLException &e) {
    ..
}
```

The debug control string "d:t:O,/tmp/client.trace" specifies what kind of debug traces to be collected.

The flag 'd' enables output from DBUG_ macros for the current state. In MySQL, common tags to print with the option 'd' are ENTER, EXIT, ERROR, WARNING, INFO, and LOOP. For the complete list of supported DBUG macros, check the DBUG C Program Debugging Package document.

The flag 't' enables function call/exit trace lines. It can be be followed by a list giving a numeric maximum trace level, beyond which no output occurs for either debugging or tracing macros. eg., t,20

The flag 'O,/tmp/client.trace' redirects the debugger output stream to the specified file, /tmp/client.trace, on the client machine. The file is flushed between each write. When needed, the file is closed and re-opened between each write.

The flag 'f' followed by an empty list of named functions disables the debug trace.

For the complete list of debug control flags and their descriptions, check the MySQL Reference Manual.

Internal Traces Generated by the Connector/C++ Driver

Connector/C++ driver's internal debug trace is a call trace. It shows each and every function call and sometimes the function arguments and other related information. However the internal tracing functionality is not available by default with Connector/C++. When building the Connector/C++ driver, make sure to enable the tracing module by using the CMake option, -DMYSQLCPPCONN_TRACE_ENABLE:BOOL=1. It is not necessary to link the application against the debug version of the MySQL Client Library to generate internal traces of Connector/C++. Detailed instructions are in the Installing MySQL Connector/C++ section of this tutorial.

Compiling the connector driver with the tracing functionality enabled will incur two additional tracing function call overhead per each driver function call as shown below.

```
...
#     <MySQL_Connection::init
#     >MySQL_Connection::setAutoCommit
#     <MySQL_Connection::setAutoCommit
Database connection's autocommit mode = #     >MySQL_Connection::getAutoCommit
Executing query: SELECT * FROM City ..
#     >MySQL_Statement::execute
#     |   INF: this=80694b8
#     |   INF: query=START TRANSACTION;
#     |   >MySQL_Statement::do_query
#     |   |   INF: this=80694b8
#     |   <MySQL_Statement::do_query
#     <MySQL_Statement::execute
#     >MySQL_Statement::execute
#     |   INF: this=80694b8
#     |   INF: query=INSERT INTO City (CityName) VALUES ('Napier, New Zealand')
#     |   >MySQL_Statement::do_query
#     |   |   INF: this=80694b8
#     |   <MySQL_Statement::do_query
#     <MySQL_Statement::execute
#     >MySQL_Statement::getUpdateCount
#     <MySQL_Statement::getUpdateCount
update count = 1
#     >MySQL_Statement::execute
#     |   INF: this=80694b8
#     |   INF: query=ROLLBACK;
#     |   >MySQL_Statement::do_query
#     |   |   INF: this=80694b8
...
```

When the Connector/C++ driver is built with the CMake option, `-DMYSQLCPPCONN_TRACE_ENABLE:BOOL=1`, users on non-Windows platforms can enable the debug tracing for the whole application by setting the environment variable `MYSQLCPPCONN_TRACE_ENABLED` to any value on the command line prior to running the C++ client application from the same shell.

```
bash# MYSQLCPPCONN_TRACE_ENABLED=1 ./mysqlconnectorc++client

#     >MySQL_Connection::init
#     |   INF: hostName=tcp://:
#     |   INF: user=root
#     |   INF: port=0
#     |   INF: schema=
#     |   INF: socket=
#     |   >MySQL_Connection::setAutoCommit
#     |   etAutoCommit
#     |   >MySQL_Connection::setTransactionIsolation
#     |   etTransactionIsolation
#     nit
...
```

Controlling the debug traces with the environment variable is simple and does not require any code instrumentation in the application. It generates the traces for all the driver specific function calls from the application. But if you want per connection debug traces or fine-grained control over the traces to be generated, consider instrumenting your code with the `Connection::setClientOptions` method. For example, if you want to see what is happening inside the driver when a driver supported function is called from the application, call `setClientOptions()` twice from your application -- once to enable the client tracing right before calling the driver specific function you are interested in, and the second call is to disable the client tracing right after calling the function.

The following code fragment demonstrates how to generate connector's internal debug trace for the `prepareStatement()` function call using `Connection::setClientOptions()`.

```
Driver *driver;
Connection *con;
PreparedStatement *prep_stmt;

string url, user, password;
int on_off = 0;

try {
    driver = get_driver_instance();

    con = driver -> connect(url, user, password);

    /* enable client tracing */
    on_off = 1;
    con -> setClientOption ("clientTrace", &on_off);

    prep_stmt = con -> prepareStatement ("INSERT INTO City (CityName) VALUES (?)");

    /* disable client tracing */
    on_off = 0;
    con -> setClientOption ("clientTrace", &on_off);

} catch (SQLException &e) {
    ..
}
```

The call to `setClientOption` method shown in the above code causes the debug trace to be written to the `stdout`. It is not required to set the environment variable `MYSQLCPPCONN_TRACE_ENABLED` in this case.

Be aware that trace enabled versions may cause higher CPU usage even if the overall run time of your application is not impacted significantly. There will be some additional I/O overhead if the debug trace has to be written to a trace file.

For More Information

- MySQL Server
  - MySQL Reference Manual
- MySQL Connector/C++
  - MySQL Connector/C++ Project Page
  - MySQL Connector/C++ Reference Manual
  - MySQL Connector/C++ : Installing from Source
  - Notes on using the MySQL Connector/C++ API
  - MySQL Connector/C++ : Filtering Debug Output
  - MySQL Connector/C++ Known Bugs and Issues. Report new bugs at bugs.mysql.com
  - MySQL Connector/C++ Change History
  - MySQL Connector/C++ Feature requests
  - MySQL Connector/C++ mailing list
  - MySQL Connector/C++ discussion forum
- MySQL Connector/C
  - MySQL Connector/C 6.0 download page
  - MySQL Connector/C Reference Manual
- NetBeans IDE
  - MySQL Connector/C++: How to Build a Client on Linux using NetBeans 6.5 (for Dummies)
  - Tutorials: Developing C and C++ Applications using NetBeans IDE
- Visual Studio IDE
  - MySQL Connector/C++: Guide on building a Windows Client using Visual Studio (for Dummies)
- JDBC
  - JDBC 4.0 API Specification
  - JDBC Basics
- Weblogs
  - Ulf Wendel's weblog
  - Andrey Hristov's weblog
  - Lance Andersen’s weblog

Acknowledgments

The author would like to acknowledge Andrey Hristov, Ulf Wendel, Lance Andersen and Edwin DeSouza of Sun | MySQL AB, for the extensive feedback on this article.

About the Author

Giri Mandalika is a software engineer in Sun's ISV Engineering organization. Giri works with partners and ISVs to make Sun the preferred vendor of choice for deploying applications. Currently, Giri is focused on standard benchmarks, optimization, and scalability of enterprise applications on Sun platforms.

Appendix I: Installing MySQL Connector/C++ from Source

Before you proceed with the installation from source, check the Connector/C++ downloads page to see whether the driver is available in binary form for your build platform, and whether it works for you. In general, all the pre-compiled binaries are built with the best possible options for all the supported platforms. Installation from the binary package is very straight forward on all platforms - simply unpacking the zip archive in a desired location installs the Connector/C++ driver.

Contents

These instructions outline the dependencies and the steps to build and install the Connector/C++ driver on all supported platforms. bash shell was used to show all the examples in this tutorial.

Build-time Dependencies

- CMake 2.6.2 or later on Microsoft Windows, and version 2.4.2 or later on rest of the supported platforms

  CMake is a cross-platform build system that generates native build files for your environment.
- MySQL client programming support

  Connector/C++ is based on the MySQL C API, and the driver is linked against the MySQL Client Library.
- GLib 2.2.3 or later on Linux

Install CMake

Check the CMake software downloads page for the pre-compiled binary for your build platform. If the binary is not available, or if you plan to build CMake from a source tree, checkInstalling CMake page for the instructions to build CMake.

The following example demonstrates the steps involved in building & installing CMake in a non-default location on Solaris operating system. Sun Studio C/C++ compilers were used to build CMake.

```
# gunzip -c cmake-2.6.3.tar.gz | tar -xvf -
# cd cmake-2.6.3

# export CC=cc
# export CXX=CC

# ./bootstrap --prefix=/export/expts/CMake

# make
# make install

# export PATH=/export/expts/CMake/bin:$PATH

# which cmake
/export/expts/CMake/bin/cmake

# cmake -version
cmake version 2.6-patch 3
```

Install MySQL Client Programming Support

MySQL Connector/C++ mandates the installation of MySQL 5.1 [or later] client programming support -- MySQL specific libraries and the header files. Majority of the MySQL Server installation methods on different platforms install the necessary files by default. However if you installed MySQL from RPM files on Linux, be sure that you've installed the developer RPM. The client programs are in the client RPM, but client programming support is in the developer RPM.

If you do not have the MySQL Server installed and if you only need the MySQL Client Library along with the header files, consider installing MySQL Connector/C on the build machine for the required client programming support. MySQL Connector/C is a C client library for client-server communication. It is a standalone replacement for the MySQL Client Library shipped with the MySQL Server. As of this writing, MySQL Connector/C 6.0 is available in the source form as well as the binary form for Solaris, Linux, Windows, FreeBSD, Mac OS X, HP-UX and AIX platforms. MSI installer and a binary zip file without the installer is available for Windows, where as the binary package is available as compressed GNU TAR archive (tar.gz) for the rest of the platforms. You can download the source code and the pre-compiled binary from the Connector/C 6.0 download page. Building the Connector/C driver from the source code is very similar to building the Connector/C++ driver from the source code.

If Connector/C does not fit your requirements, download MySQL, then install.

Add the location of the MySQL Client Library to the search path that the runtime linker uses to find dynamic dependencies. For example, on Solaris OS, use LD_LIBRARY_PATH environment variable to set the runtime linker search path as shown below.

```
Connector/Cexport LD_LIBRARY_PATH=/export/expts/MySQLConnectorC6.0/lib:$LD_LIBRARY_PATH
```

= OR =

```
MySQL Server

# ls -1 /opt/coolstack/mysql_32bit/lib/mysql/libmysqlclient_r.so.16.0.0
/opt/coolstack/mysql_32bit/lib/mysql/libmysqlclient_r.so.16.0.0

# export LD_LIBRARY_PATH=/opt/coolstack/mysql_32bit/lib/mysql:$LD_LIBRARY_PATH

# ls -1 /export/expts/MySQLConnectorC6.0/lib/libmysqlclient_r.so
/export/expts/MySQLConnectorC6.0/lib/libmysqlclient_r.so

#
```

Build & Install MySQL Connector/C++

The following step-by-step instructions are applicable to all supported UNIX like platforms. Check MySQL Forge Wiki and MySQL Connector/C++: Guide on building a Windows Client using Visual Studio (for Dummies) pages for the Windows build instructions.

1. Download the latest version of the MySQL Connector/C++ source tree from dev.mysql.com.
2. Extract the source files.

   # gunzip -c mysql-connector-c++-1.0.4-beta.tar.gz | tar -xvf -

   # cd mysql-connector-c++-1.0.4-beta
3. If you are not using the GNU C/C++ compilers, you need to tell the Make build system which compiler to use by setting the environment variables `CC` and `CXX`.

   For example, to use Sun Studio C/C++ compilers on Solaris and Linux platforms, set the environment variables `CC` and `CXX` as shown below.

   ```
   # which cc
   /usr/bin/cc

   # which CC
   /usr/bin/CC

   # export CC=cc
   # export CXX=CC
   ```

4. Run CMake, the cross-platform build system, to generate the `Makefile`. In this step, you can specify the compiler and link-editor options. In addition, you can specify a variety of options like the installation directory, build type - debug or non-debug, location of `mysql_config` in this step. For all the supported options and the exact syntax, check theCMake Documentation.

   ```
   # cmake -DCMAKE_REQUIRED_FLAGS=-xO4 -DCMAKE_INSTALL_PREFIX=/export/expts/
   MySQLConnectorC++
   ```

   On non-Windows systems, CMake tries to locate `mysql_config` file in the default locations, if the CMake variable `MYSQL_CONFIG_EXECUTABLE` is not set. If the MySQL Server is installed in a non-default location, set the CMake variable `MYSQL_CONFIG_EXECUTABLE`.

   ```
   # other options ..]cmake -DMYSQL_CONFIG_EXECUTABLE=/path/to/my/mysql/server/bin/mysql_config [..
   ```

On Windows platform, `mysql_config` is not available. Hence CMake attempts to retrieve the location of MySQL from the environment variable `$ENV{MYSQL_DIR}`. If `MYSQL_DIR` variable is not set, then CMake proceeds to check for MySQL in the following locations: `$ENV{ProgramFiles}/MySQL/*/include` and `$ENV{SystemDrive}/MySQL/*/include`.

To build debug version of the MySQL Connector/C++, set the CMake variable `CMAKE_BUILD_TYPE` to `Debug`.

```
# cmake -DCMAKE_BUILD_TYPE=Debug [.. other options ..]
```

The MySQL Connector/C++ driver is capable of generating debug traces. By default, connector's internal tracing functionality is not available. If you need it, you have to enable the tracing module at compile time using the CMake option, `-DMYSQLCPPCONN_TRACE_ENABLE:BOOL=1`. The tracing functionality is available with both debug and non-debug builds of the connector, hence it is not required to build debug version of the MySQL Connector/C++ driver in order to use the internal tracing functionality of the connector.

To enable the tracing module, set the CMake variable `MYSQLCPPCONN_TRACE_ENABLE` with the type `BOOL` and the value set to 1.

```
# cmake -DMYSQLCPPCONN_TRACE_ENABLE:BOOL=1 [.. other options ..]
```

In addition to the internal debug traces, Connector/C++ can also generate debug traces when linked against the debug version of MySQL Client Library. Check Debug Tracing with MySQL Connector/C++ for more details.

Run `cmake -L` to check the list of options being used by CMake.

5. Run `make` to build the Connector/C++ driver.

```
# make
```

6. Finally run `make` with the `install` option to install the Connector/C++ driver in the designated location.

```
# make install
..
-- Installing: /export/expts/MySQLConnectorC++/lib/libmysqlcppconn.so.1
-- Installing: /export/expts/MySQLConnectorC++/lib/libmysqlcppconn-static.a
-- Installing: /export/expts/MySQLConnectorC++/include/mysql_connection.h
-- Installing: /export/expts/MySQLConnectorC++/include/mysql_driver.h
..
```

If the CMake variable `CMAKE_INSTALL_PREFIX` was not set during the `Makefile` generation (step #4), the driver along with the header files will be installed in the default location, `/usr/local`.