MySQL Connector/C++

MySQL Connector/C++

Abstract

This manual describes MySQL Connector/C++, the C++ interface for communicating with MySQL servers.

Document generated on: 2009-06-02 (revision: 15165)

For more information on the terms of this license, for details on how the MySQL documentation is built and produced, or if you are interested in doing a translation, please contact the Documentation Team.

For additional licensing information, including licenses for libraries used by MySQL, see Preface, Notes, Licenses.

If you want help with using MySQL, please visit either the MySQL Forums or MySQL Mailing Lists where you can discuss your issues with other MySQL users.

For additional documentation on MySQL products, including translations of the documentation into other languages, and downloadable versions in variety of formats, including HTML, CHM, and PDF formats, see MySQL Documentation Library.

MySQL Connector/C++

> **Caution**
>
> Please note, official support is not available for the beta version of MySQL Connector/C++.

MySQL Connector/C++ is a MySQL database connector for C++.

The MySQL Connector/C++ is licensed under the terms of the GPL, like most MySQL Connectors. There are special exceptions to the terms and conditions of the GPL as it is applied to this software, see FLOSS License Exception. If you need a non-GPL license for commercial distribution please contact us.

The MySQL Connector/C++ is compatible with the JDBC 4.0 API. However, MySQL Connector/C++ does not implement all of the JDBC 4.0 API. The MySQL Connector/C++ current version features the following classes:

- `Connection`

- `DatabaseMetaData`

- `Driver`

- `PreparedStatement`

- `ResultSet`

- `ResultSetMetaData`

- `Savepoint`

- `Statement`

The JDBC 4.0 API defines approximately 450 methods for the above mentioned classes. MySQL Connector/C++ implements around 80% of these and makes them available in the preview release.

The Beta release has been successfully compiled and tested on the following platforms:

AIX

- 5.2 (PPC32, PPC64)

- 5.3 (PPC32, PPC64)

FreeBSD

- 6.0 (x86, x86_64)

HPUX

- 11.11 (PA-RISC 32bit, PA-RISC 64bit)

Linux

- Debian 3.1 (PPC32, x86)

- FC4 (x86)

- RHEL 3 (ia64, x86, x86_64)

- RHEL 4 (ia64, x86, x86_64)

- RHEL 5 (ia64, x86, x86_64)

- SLES 9 (ia64, x86, x86_64)

- SLES 10 (ia64, x86_64)

- SuSE 10.3, (x86_64)

- Ubuntu 8.04 (x86)

- Ubuntu 8.10 (x86_64)

Mac

- MacOSX 10.3 (PPC32, PPC64)

- MacOSX 10.4 (PPC32, PPC64, x86)

- MacOSX 10.5 (PPC32, PPC64, x86, x86_64)

SunOS

- Solaris 8 (SPARC32, SPARC64, x86)

- Solaris 9 (SPARC32, SPARC64, x86)

- Solaris 10 (SPARC32, SPARC64, x86, x86_64)

Windows

- XP Professional (32bit)

- 2003 (64bit)

Future versions will run on all platforms supported by the MySQL Server.

> **Note**
>
> MySQL Connector/C++ supports MySQL 5.1 and later.

> **Note**
>
> MySQL Connector/C++ supports only Microsoft Visual Studio 2003 and above on Windows.

MySQL Connector/C++ Download

You can download the source code for the MySQL Connector/C++ preview release at the download site.

MySQL Connector/C++ Source repository

The latest development version is also available through Launchpad.

Bazaar is used for the MySQL Connector/C++ code repository. You can check out the source code using the `bzr` command line tool:

```
shell> bzr branch lp:~mysql/mysql-connector-cpp/trunk .
```

The source of the 1.0.3alpha release is available from the following branch:

```
shell> bzr branch lp:~andrey-mysql/mysql-connector-cpp/v1_0_3
```

The source of the 1.0.2alpha release is available from the following branch:

```
shell> bzr branch lp:~andrey-mysql/mysql-connector-cpp/v1_0_2
```

The source of the 1.0.1alpha release is available from the following branch:

```
shell> bzr branch lp:~andrey-mysql/mysql-connector-cpp/v1_0_1
```

Binary distributions

Starting with 1.0.4 Beta, binary distributions will be made available in addition to source code releases. The releases available are shown below.

Microsoft Windows platform:

- Without installer, a Zip file

- MSI installer package

Other platforms:

- Compressed GNU TAR archive (tar.gz)

> **Note**
>
> Note that source packages are available for all platforms in the Compressed GNU TAR archive (tar.gz) format.

MySQL Connector/C++ Advantages

Using MySQL Connector/C++ instead of the MySQL C API (MySQL Client Library) offers the following advantages for C++ users:

- Convenience of pure C++, no C function calls required

- Supports an industry standard API, JDBC 4.0

- Supports the object-oriented programming paradigm

- Reduces development time

- MySQL Connector/C++ is licensed under the GPL with the FLOSS License Exception

- MySQL Connector/C++ is available under a commercial license upon request

MySQL Connector/C++ Status

MySQL Connector/C++ is available as a development preview version. We kindly ask users and developers to try it out and provide us with feedback. We do not encourage you to use it in production environments.

Note that MySQL Workbench is successfully using MySQL Connector/C++.

> **Note**
>
> Sun Microsystems does not provide formal support for MySQL Connector/C++.

If you have any queries please contact us.

Chapter 1. MySQL Connector/C++ Source Installation

> **Caution**
>
> Please note, official support is not available for the beta version of MySQL Connector/C++.

The MySQL Connector/C++ is based on the MySQL Client Library (MySQL C API). MySQL Connector/C++ is linked against the MySQL Client Library. You need to have the MySQL Client Library installed in order to compile MySQL Connector/C++.

You also need to have the cross-platform build tool CMake 2.4, or newer, and GLib 2.2.3 or newer installed. Check the README file included with the distribution for platform specific notes on building for Windows and SunOS.

Typically the MySQL Client Library is installed when the MySQL Server is installed. However, check your operating system documentation for other installation options.

1.1. Building source on Unix, Solaris and Mac OS X

1. Run CMake to build a Makefile:

    ```
    shell> me@host:/path/to/mysql-connector-cpp> cmake .
    -- Check for working C compiler: /usr/local/bin/gcc
    -- Check for working C compiler: /usr/local/bin/gcc -- works
    [...]
    -- Generating done
    -- Build files have been written to: /path/to/mysql-connector-cpp/
    ```

    On non-Windows systems, CMake first checks to see if the CMake variable `MYSQL_CONFIG_EXECUTABLE` is set. If it is not found CMake will try to locate `mysql_config` in the default locations.

    If you have any problems with the configure process please check the troubleshooting instructions below.

2. Use make to build the libraries. First make sure you have a clean build:

    ```
    shell> me@host:/path/to/mysql-connector-cpp> make clean
    ```

    Then build the connector:

    ```
    me@host:/path/to/mysql-connector-cpp> make
    [  1%] Building CXX object »
    driver/CMakeFiles/mysqlcppconn.dir/mysql_connection.o
    [  3%] Building CXX object »
    driver/CMakeFiles/mysqlcppconn.dir/mysql_constructed_resultset.o
    [...]
    [100%] Building CXX object examples/CMakeFiles/statement.dir/statement.o
    Linking CXX executable statement
    ```

    If all goes well, you will find the MySQL Connector/C++ library in `/path/to/cppconn/libmysqlcppconn.so`.

3. Finally make sure the header and library files are installed to their correct locations:

    ```
    make install
    ```

    Unless you have changed this in the configuration step, the header files will be copied to the directory `/usr/local/include`. The header files copied are `mysql_connection.h` and `mysql_driver.h`.

    Again, unless you have specified otherwise, the library files will be copied to `/usr/local/lib`. The files copied are `libmysqlcppcon.so`, the dynamic library, and `libmysqlcppconn-static.a`, the static library.

If you encounter any errors, please first carry out the checks shown below:

1. CMake options: MySQL installation path, debug version and more

    In case of configure and/or compile problems check the list of CMake options:

    ```
    shell> me@host:/path/to/mysql-connector-cpp> cmake -L
    [...]
    CMAKE_BACKWARDS_COMPATIBILITY:STRING=2.4
    CMAKE_BUILD_TYPE:STRING=
    CMAKE_INSTALL_PREFIX:PATH=/usr/local
    EXECUTABLE_OUTPUT_PATH:PATH=
    ```

```
LIBRARY_OUTPUT_PATH:PATH=
MYSQLCPPCONN_GCOV_ENABLE:BOOL=0
MYSQLCPPCONN_TRACE_ENABLE:BOOL=0
MYSQL_CONFIG_EXECUTABLE:FILEPATH=/usr/bin/mysql_config
```

For example, if your MySQL Server installation path is not `/usr/local/mysql` and you want to build a debug version of the MySQL Connector/C++ use:

```
shell> me@host:/path/to/mysql-connector-cpp> cmake »
-D CMAKE_BUILD_TYPE:STRING=Debug »
-D MYSQL_CONFIG_EXECUTABLE=/path/to/my/mysql/server/bin/mysql_config .
```

2. Verify your settings with `cmake -L`:

```
shell> me@host:/path/to/mysql-connector-cpp> cmake -L
[...]
CMAKE_BACKWARDS_COMPATIBILITY:STRING=2.4
CMAKE_BUILD_TYPE:STRING=
CMAKE_INSTALL_PREFIX:PATH=/usr/local
EXECUTABLE_OUTPUT_PATH:PATH=
LIBRARY_OUTPUT_PATH:PATH=
MYSQLCPPCONN_GCOV_ENABLE:BOOL=0
MYSQLCPPCONN_TRACE_ENABLE:BOOL=0
MYSQL_CONFIG_EXECUTABLE=/path/to/my/mysql/server/bin/mysql_config
```

Proceed by carrying out a `make clean` command followed by a `make` command, as described above.

1.2. Building on Windows

> **Note**
>
> Please note the only compiler formally supported for Windows is Microsoft Visual Studio 2003 and above.

The basic steps for building the connector on Windows are the same as for Unix. It is important to use CMake 2.6.2 or newer to generate build files for your compiler and to invoke the compiler.

> **Note**
>
> On Windows, `mysql_config` is not present, so CMake will attempt to retrieve the location of MySQL from the environment variable `$ENV{MYSQL_DIR}`. If `MYSQL_DIR` is not set, CMake will then proceed to check for MySQL in the following locations: `$ENV{ProgramFiles}/MySQL/*/include`, and `$ENV{SystemDrive}/MySQL/*/include`.

CMake makes it easy for you to try out other compilers. However, you may experience compile warnings, compile errors or linking issues not detected by Visual Studio. Patches are gratefully accepted to fix issues with other compilers.

Consult the CMake manual or check `cmake --help` to find out which build systems are supported by your CMake version:

```
C:\>cmake --help
cmake version 2.6-patch 2
Usage
[...]
Generators
The following generators are available on this platform:
  Borland Makefiles            = Generates Borland makefiles.
  MSYS Makefiles               = Generates MSYS makefiles.
  MinGW Makefiles              = Generates a make file for use with
                                 mingw32-make.
  NMake Makefiles              = Generates NMake makefiles.
  Unix Makefiles               = Generates standard UNIX makefiles.
  Visual Studio 6              = Generates Visual Studio 6 project files.
  Visual Studio 7              = Generates Visual Studio .NET 2002 project
                                 files.
  Visual Studio 7 .NET 2003    = Generates Visual Studio .NET 2003 project
                                 files.
  Visual Studio 8 2005         = Generates Visual Studio .NET 2005 project
                                 files.
  Visual Studio 8 2005 Win64   = Generates Visual Studio .NET 2005 Win64
                                 project files.
  Visual Studio 9 2008         = Generates Visual Studio 9 2008 project fil
  Visual Studio 9 2008 Win64   = Generates Visual Studio 9 2008 Win64 proje
                                 files.
[...]
```

It is likely that your CMake binary will support more compilers, known by CMake as *generators*, than supported by MySQL Connector/C++. We have built the connector using the following generators:

- Microsoft Visual Studio 8 (Visual Studio 2005)

- Microsoft Visual Studio 9 (Visual Studio 2008, Visual Studio 2008 Express)

- NMake

Please see the building instructions for Unix, Solaris and Mac OS X for troubleshooting and configuration hints.

The steps to build the connector are given below:

1. Run CMake to generate build files for your *generator*:

   Visual Studio

```
 C:\path_to_mysql_cpp>cmake -G "Visual Studio 9 2008"
-- Check for working C compiler: cl
-- Check for working C compiler: cl -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: cl
-- Check for working CXX compiler: cl -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- ENV{MYSQL_DIR} =
-- MySQL Include dir: C:/Programme/MySQL/MySQL Server 5.1/include
-- MySQL Library    : C:/Progams/MySQL/MySQL Server 5.1/lib/opt/mysqlclient.lib
-- MySQL Library dir: C:/Progams/MySQL/MySQL Server 5.1/lib/opt
-- MySQL CFLAGS:
-- MySQL Link flags:
-- MySQL Include dir: C:/Progams/MySQL/MySQL Server 5.1/include
-- MySQL Library dir: C:/Progams/MySQL/MySQL Server 5.1/lib/opt
-- MySQL CFLAGS:
-- MySQL Link flags:
-- Configuring cppconn
-- Configuring test cases
-- Looking for isinf
-- Looking for isinf - not found
-- Looking for isinf
-- Looking for isinf - not found.
-- Looking for finite
-- Looking for finite - not found.
-- Configuring C/J junit tests port
-- Configuring examples
-- Configuring done
-- Generating done
-- Build files have been written to: C:\path_to_mysql_cpp
C:\path_to_mysql_cpp>dir *.sln *.vcproj
[...]
19.11.2008  12:16            23.332 MYSQLCPPCONN.sln
[...]
19.11.2008  12:16            27.564 ALL_BUILD.vcproj
19.11.2008  12:16            27.869 INSTALL.vcproj
19.11.2008  12:16            28.073 PACKAGE.vcproj
19.11.2008  12:16            27.495 ZERO_CHECK.vcproj
```

   NMake

```
 C:\path_to_mysql_cpp>cmake -G "NMake Makefiles"
-- The C compiler identification is MSVC
-- The CXX compiler identification is MSVC
[...]
-- Build files have been written to: C:\path_to_mysql_cpp
```

2. Use your compiler to build MySQL Connector/C++

   Visual Studio - GUI

   Open the newly generated project files in the Visual Studio GUI or use a Visual Studio command line to build the driver. The project files contain a variety of different configurations. Among them debug and non-debug versions.

   Visual Studio - NMake

```
C:\path_to_mysql_cpp>nmake
Microsoft (R) Program Maintenance Utility Version 9.00.30729.01
Copyright (C) Microsoft Corporation.  All rights reserved.
Scanning dependencies of target mysqlcppconn
[  2%] Building CXX object driver/CMakeFiles/mysqlcppconn.dir/mysql_connection.obj
mysql_connection.cpp
[...]
Linking CXX executable statement.exe
[100%] Built target statement
```

Chapter 2. MySQL Connector/C++ Binary Installation

Caution

Please note, official support is not available for the beta version of MySQL Connector/C++.

Caution

One problem that can occur is when the tools you use to build your application are not compatible with the tools used to build the binary versions of MySQL Connector/C++. Ideally you need to build your application with the same tools that were used to build the MySQL Connector/C++ binaries. To help with this the following resources are provided.

All distributions contain a `README` file, which contains platform-specific notes. At the end of the `README` file contained in the binary distribution you will find the settings used to build the binaries. If you experience build-related issues on a platform, it may help to check the settings used on the platform to build the binary.

For your convenience the same information, but more frequently updated, can be found on the MySQL Forge site.

A better solution is to build your MySQL Connector/C++ libraries from the source code, using the same tools that you use for building your application. This ensures compatibility.

Archive Package

Unpack the archive into an appropriate directory. If you plan to use a dynamically linked version of MySQL Connector/C++, make sure that your system can reference the MySQL Client Library. Consult your operating system documentation on how do modify and expand the search path for libraries. In case you cannot modify the library search path it may help to copy your application, the MySQL Connector/C++ library and the MySQL Client Library into the same directory. Most systems search for libraries in the current directory.

Windows MSI Installer

Windows users can choose between two binary packages:

1. Without installer (unzip in C:\)

2. Windows MSI Installer (x86)

Using the MSI Installer may be the easiest solution. Running the MSI Installer does not require any administrative permissions as it simply copies files.

Figure 2.1. Windows Installer Welcome Screen

Figure 2.2. Windows Installer Overview Screen
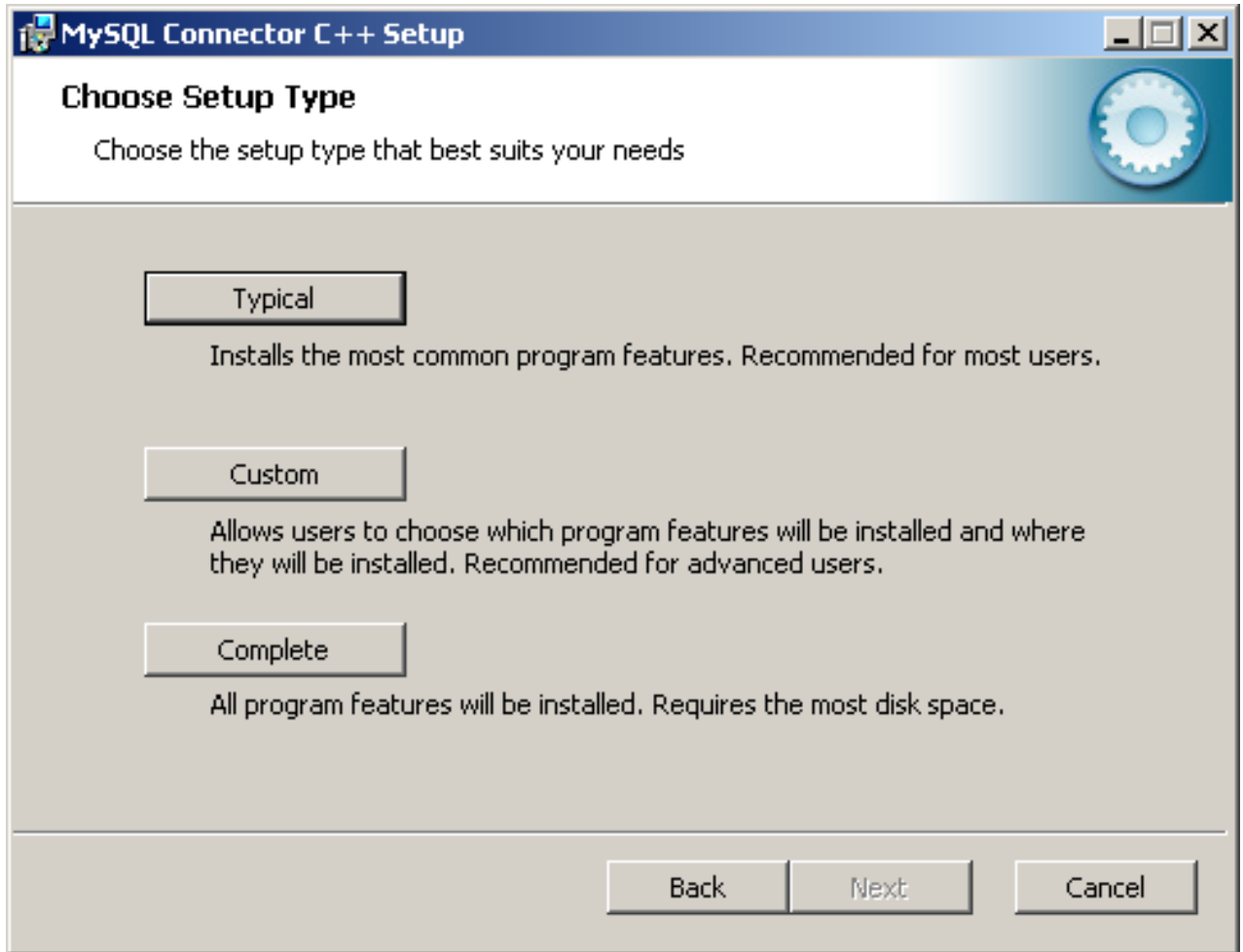
The "Typical" installation consists of all required header files and the Release libraries. The only available "Custom" installation option allows you to install additional Debug versions of the connector libraries.

Figure 2.3. Windows Installer Custom Setup Screen
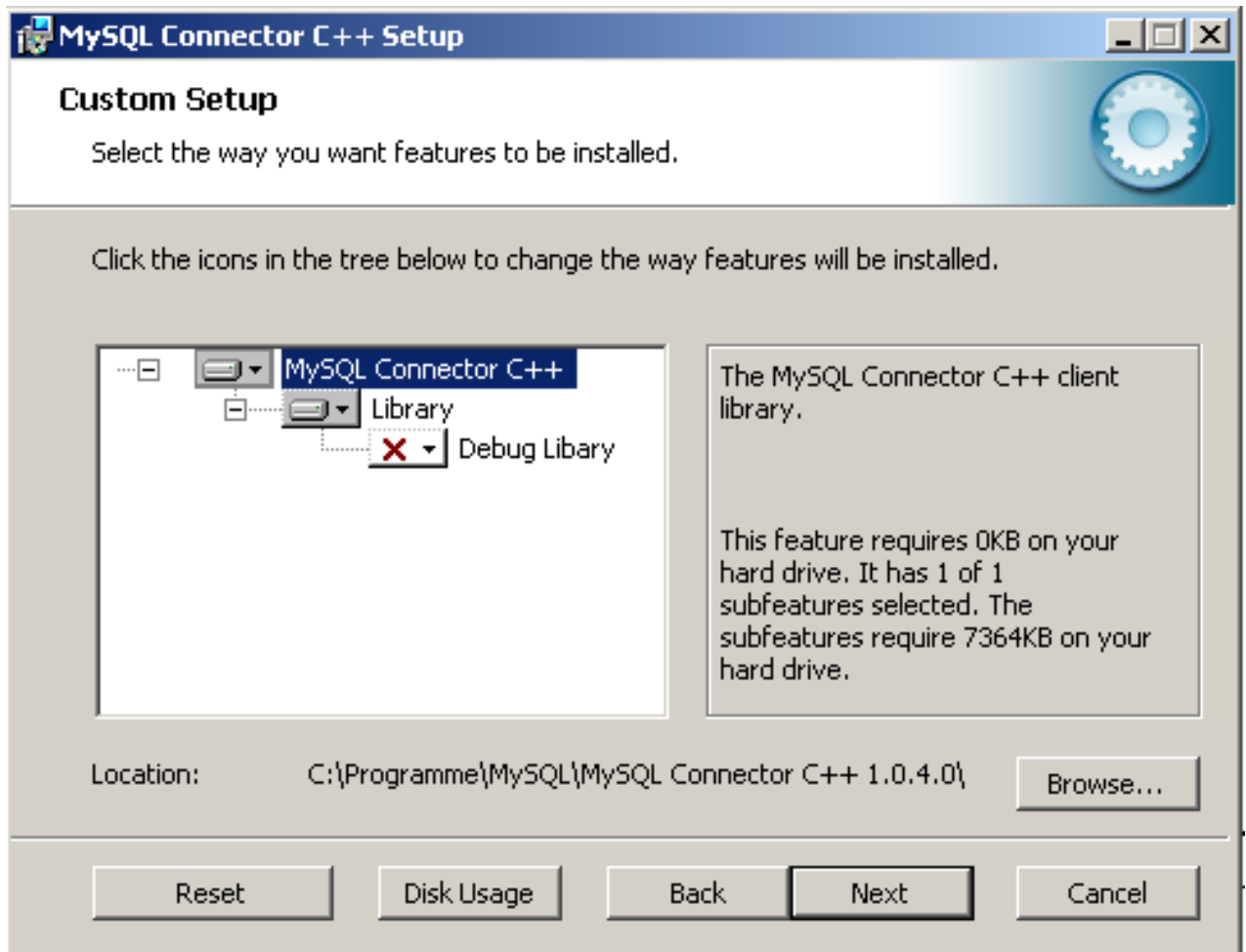
Chapter 3. MySQL Connector/C++ Getting Started: Usage Examples

> **Caution**
>
> Please note, official support is not available for the beta version of MySQL Connector/C++.

The download package contains usage examples in the directory `examples/`. The examples explain the basic usage of the following classes:

- `Connection`

- `Driver`

- `PreparedStatement`

- `ResultSet`

- `ResultSetMetaData`

- `Statement`

The examples cover:

- Using the `Driver` class to connect to MySQL

- Creating tables, inserting rows, fetching rows using (simple) statements

- Creating tables, inserting rows, fetching rows using prepared statements

- Hints for working around prepared statement limitations

- Accessing result set meta data

The examples in this document are only code snippets. The code snippets provide a brief overview on the API. They are not complete programs. Please check the `examples/` directory of your MySQL Connector/C++ installation for complete programs. Please also read the `README` file in the `examples/` directory. Note to test the example code you will first need to edit the `examples.h` file in the `examples/` directory, to add your connection information. Then simply rebuild the code by issuing a `make` command.

The examples in the `examples/` directory include:

- `examples/connect.cpp`:

  How to create a connection, insert data into MySQL and handle exceptions.

- `examples/connection_meta_schemaobj.cpp`:

  How to obtain meta data associated with a connection object, for example, a list of tables, databases, MySQL version, connector version.

- `examples/debug.cpp`:

  How to activate and deactivate the MySQL Connector/C++ debug protocol.

- `examples/exceptions.cpp`:

  A closer look at the exceptions thrown by the connector and how to fetch error information.

- `examples/prepared_statements.cpp`:

  How to run Prepared Statements including an example how to handle SQL commands that cannot be prepared by the MySQL Server.

- `examples/resultset.cpp`:

How to fetch data and iterate over the result set (cursor).

- `examples/resultset_meta.cpp`:

  How to obtain meta data associated with a result set, for example, number of columns and column types.

- `examples/resultset_types.cpp`:

  Result sets returned from meta data methods - this is more a test than much of an example.

- `examples/standalone_example.cpp`:

  Simple standalone program not integrated into regular CMake builds.

- `examples/statements.cpp`:

  How to run SQL commands without using Prepared Statements.

- `examples/cpp_trace_analyzer.cpp`:

  This example shows how to filter the output of the debug trace. Please see the inline comments for further documentation. This script is unsupported.

3.1. MySQL Connector/C++ Connecting to MySQL

A connection to MySQL is established by retrieving an instance of `sql::Connection` from a `sql::mysql::MySQL_Driver` object. A `sql::mysql::MySQL_Driver` object is returned by `sql::mysql::MySQL_Driver::get_mysql_driver_instance()`.

```
sql::mysql::MySQL_Driver *driver;
sql::Connection *con;
driver = sql::mysql::MySQL_Driver::get_mysql_driver_instance();
con = driver->connect("tcp://127.0.0.1:3306", "user", "password");
delete con;
```

Make sure that you free the `sql::Connection` object as soon as you do not need it any more. But do not explicitly free the connector object!

3.2. MySQL Connector/C++ Running a simple query

For running simple queries you can use the methods `sql::Statement::execute()`, `sql::Statement::executeQuery()` and `sql::Statement::executeUpdate()`. The method `sql::Statement::execute()` should be used if your query does not return a result set or if your query returns more than one result set. See the `examples/` directory for more on this.

```
sql::mysql::MySQL_Driver *driver;
sql::Connection *con;
sql::Statement *stmt
driver = sql::mysql::get_mysql_driver_instance();
con = driver->connect("tcp://127.0.0.1:3306", "user", "password");
stmt = con->createStatement();
stmt->execute("USE " EXAMPLE_DB);
stmt->execute("DROP TABLE IF EXISTS test");
stmt->execute("CREATE TABLE test(id INT, label CHAR(1))");
stmt->execute("INSERT INTO test(id, label) VALUES (1, 'a')");
delete stmt;
delete con;
```

Note that you have to free `sql::Statement` and `sql::Connection` objects explicitly using delete.

3.3. MySQL Connector/C++ Fetching results

The API for fetching result sets is identical for (simple) statments and prepared statements. If your query returns one result set you should use `sql::Statement::executeQuery()` or `sql::PreparedStatement::executeQuery()` to run your query. Both methods return `sql::ResultSet` objects. The preview version does buffer all result sets on the client to support cursors.

```
// ...
sql::Connection *con;
sql::Statement *stmt
sql::ResultSet  *res;
```

```
// ...
stmt = con->createStatement();
// ...
res = stmt->executeQuery("SELECT id, label FROM test ORDER BY id ASC");
while (res->next()) {
  // You can use either numeric offsets...
  cout << "id = " <&;t; res->getInt(0);
  // ... or column names for accessing results. »
    The latter is recommended.
  cout << ", label = '" << »
    res->getString("label") << "'" << endl;
}
delete res;
delete stmt;
delete con;
```

Note that you have to free `sql::Statement`, `sql::Connection` and `sql::ResultSet` objects explicitly using delete.

The usage of cursors is demonstrated in the examples contained in the download package.

3.4. MySQL Connector/C++ Using Prepared Statements

If you are not familiar with Prepared Statements on MySQL have an extra look at the source code comments and explanations in the file `examples/prepared_statement.cpp`.

`sql::PreparedStatement` is created by passing a SQL query to `sql::Connection::prepareStatement()`. As `sql::PreparedStatement` is derived from `sql::Statement`, you will feel familiar with the API once you have learned how to use (simple) statements (`sql::Statement`). For example, the syntax for fetching results is identical.

```
// ...
sql::Connection *con;
sql::PreparedStatement *prep_stmt
// ...
prep_stmt = con->prepareStatement("INSERT INTO test(id, label) VALUES (?, ?)");
prep_stmt->setInt(1, 1);
prep_stmt->setString(2, "a");
prep_stmt->execute();
prep_stmt->setInt(1, 2);
prep_stmt->setString(2, "b");
prep_stmt->execute();
delete prep_stmt;
delete con;
```

As usual, you have to free `sql::PreparedStatement` and `sql::Connection` objects explicitly.

3.5. MySQL Connector/C++ Complete Example 1

The following code shows a complete example of how to use MySQL Connector/C++:

```
/* Copyright 2008 Sun Microsystems, Inc.
This program is free software; you can redistribute it and/or modify
it under only the terms of the GNU General Public License as published by
the Free Software Foundation; version 2 of the License.
There are special exceptions to the terms and conditions of the GPL
as it is applied to this software. View the full text of the
exception in file EXCEPTIONS-CONNECTOR-C++ in the directory of this
software distribution.
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.
You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
*/
/* Standard C++ includes */
#include <stdlib.h>
#include <iostream>
/*
  Include directly the different
  headers from cppconn/ and mysql_driver.h + mysql_util.h
  (and mysql_connection.h). This will reduce your build time!
*/
#include "mysql_connection.h"
#include <cppconn/driver.h>
#include <cppconn/exception.h>
#include <cppconn/resultset.h>
#include <cppconn/statement.h>
using namespace std;
int main(void)
{
cout << endl;
cout << "Running 'SELECT 'Hello World!' »
    AS _message'..." << endl;
try {
```

```
    sql::Driver *driver;
    sql::Connection *con;
    sql::Statement *stmt;
    sql::ResultSet *res;
    /* Create a connection */
    driver = get_driver_instance();
    con = driver->connect("tcp://127.0.0.1:3306", "root", "root");
    /* Connect to the MySQL test database */
    con->setSchema("test");
    stmt = con->createStatement();
    res = stmt->executeQuery("SELECT 'Hello World!' AS _message");
    while (res->next()) {
      cout << "\t... MySQL replies: ";
      /* Access column data by alias or column name */
      cout << res->getString("_message") << endl;
      cout << "\t... MySQL says it again: ";
      /* Access column fata by numeric offset, 1 is the first column */
      cout << res->getString(1) << endl;
    }
    delete res;
    delete stmt;
    delete con;
} catch (sql::SQLException &e) {
    cout << "# ERR: SQLException in " << __FILE__;
    cout << "(" << __FUNCTION__ << ") on line " »
       << __LINE__ << endl;
    cout << "# ERR: " << e.what();
    cout << " (MySQL error code: " << e.getErrorCode();
    cout << ", SQLState: " << e.getSQLState() << " )" << endl;
}
cout << endl;
return EXIT_SUCCESS;
}
```

3.6. MySQL Connector/C++ Complete Example 2

The following code shows a complete example of how to use MySQL Connector/C++:

```
/* Copyright 2008 Sun Microsystems, Inc.
This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; version 2 of the License.
There are special exceptions to the terms and conditions of the GPL
as it is applied to this software. View the full text of the
exception in file EXCEPTIONS-CONNECTOR-C++ in the directory of this
software distribution.
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.
You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
*/
/* Standard C++ includes */
#include <stdlib.h>
#include <iostream>
/*
  Include directly the different
  headers from cppconn/ and mysql_driver.h + mysql_util.h
  (and mysql_connection.h). This will reduce your build time!
*/
#include "mysql_connection.h"
#include <cppconn/driver.h>
#include <cppconn/exception.h>
#include <cppconn/resultset.h>
#include <cppconn/statement.h>
#include <cppconn/prepared_statement.h>
using namespace std;
int main(void)
{
cout << endl;
cout << "Let's have MySQL count from 10 to 1..." << endl;
try {
    sql::Driver *driver;
    sql::Connection *con;
    sql::Statement *stmt;
    sql::ResultSet *res;
    sql::PreparedStatement *pstmt;
    /* Create a connection */
    driver = get_driver_instance();
    con = driver->connect("tcp://127.0.0.1:3306", "root", "root");
    /* Connect to the MySQL test database */
    con->setSchema("test");
    stmt = con->createStatement();
    stmt->execute("DROP TABLE IF EXISTS test");
    stmt->execute("CREATE TABLE test(id INT)");
    delete stmt;
    /* '?' is the supported placeholder syntax */
    pstmt = con->prepareStatement("INSERT INTO test(id) VALUES (?)");
    for (int i = 1; i <= 10; i++) {
      pstmt->setInt(1, i);
      pstmt->executeUpdate();
```

```
  }
  delete pstmt;
  /* Select in ascending order */
  pstmt = con->prepareStatement("SELECT id FROM test ORDER BY id ASC");
  res = pstmt->executeQuery();
  /* Fetch in reverse = descending order! */
  res->afterLast();
  while (res->previous())
    cout << "\t... MySQL counts: " << res->getInt("id") << endl;
  delete res;
  delete pstmt;
  delete con;
} catch (sql::SQLException &e) {
  cout << "# ERR: SQLException in " << __FILE__;
  cout << "(" << __FUNCTION__ << ") on line " »
    << __LINE__ << endl;
  cout << "# ERR: " << e.what();
  cout << " (MySQL error code: " << e.getErrorCode();
  cout << ", SQLState: " << e.getSQLState() << »
    " )" << endl;
}
cout << endl;
return EXIT_SUCCESS;
}
```

Chapter 4. MySQL Connector/C++ Debug Tracing

> **Caution**
>
> Please note, official support is not available for the beta version of MySQL Connector/C++.

Although a debugger can be used to debug your application, you may find it beneficial to turn on the debug traces of the connector. Some problems happen randomly which makes them difficult to debug using a debugger. In such cases debug traces and protocol files are more useful because they allow you to trace the activities of all instances of your program.

DTrace is a very powerful technology to trace any application without having to develop an extra trace module for your application. Unfortunately, DTrace is currently only available on OpenSolaris, Solaris, MacOS 10.5 and FreeBSD.

The MySQL Connector/C++ can write two trace files:

1. Trace file generated by the MySQL Client Library

2. Trace file generated internally by MySQL Connector/C++

The first trace file can be generated by the underlying MySQL Client Library (libmysql). To enable this trace the connector will call the C-API function `mysql_debug()` internally. Only debug versions of the MySQL Client Library are capable of writing a trace file. Therefore you need to compile MySQL Connector/C++ against a debug version of the library, if you want utilize this trace. The trace shows the internal function calls and the addresses of internal objects as you can see below:

```
>mysql_stmt_init
| >_mymalloc
| | enter: Size: 816
| | exit: ptr: 0x68e7b8
| <_mymalloc | >init_alloc_root
| | enter: root: 0x68e7b8
| | >_mymalloc
| | | enter: Size: 2064
| | | exit: ptr: 0x68eb28
[...]
```

The second trace is the MySQL Connector/C++ internal trace. It is available with debug and non-debug builds of the connector as long as you have enabled the tracing module at compile time using `cmake -DMYSQLCPPCONN_TRACE_ENABLE:BOOL=1`. By default, the tracing functionality is not available and calls to trace functions are removed by the preprocessor.

Compiling the connector with tracing functionality enabled will cause two additional tracing function calls per each connector function call. You will need to run your own benchmark to find out how much this will impact the performance of your application.

A simple test using a loop running 30,000 INSERT SQL statements showed no significant real-time impact. The two variants of this application using a trace enabled and trace disabled version of the connector performed equally well. The run time measured in real-time was not significantly impacted as long as writing a debug trace was not enabled. However, there will be a difference in the time spent in the application. When writing a debug trace the IO subsystem may become a bottleneck.

In summary, use connector builds with tracing enabled carefully. Trace enabled versions may cause higher CPU usage even if the overall run time of your application is not impacted significantly.

```
|  INF: Tracing enabled
<MySQL_Connection::setClientOption
>MySQL_Prepared_Statement::setInt
|  INF: this=0x69a2e0
| >MySQL_Prepared_Statement::checkClosed
| <MySQL_Prepared_Statement::checkClosed
<MySQL_Prepared_Statement::setInt
[...]
```

The example from `examples/debug.cpp` demonstrates how to activate the debug traces in your program. Currently they can only be activated through API calls. The traces are controlled on a per-connection basis. You can use the `setClientOptions()` method of a connection object to activate and deactivate the generation of a trace. The MySQL Client Library trace is always written into a file, whereas the connector's protocol messages are printed to standard out.

```
sql::Driver *driver;
int on_off = 1;
/* Using the Driver to create a connection */
driver = get_driver_instance();
std::auto_ptr< sql::Connection > con(driver->connect(host, user, pass));
/*
Activate debug trace of the MySQL Client Library (C-API)
Only available with a debug build of the MySQL Client Library!
*/
con->setClientOption("libmysql_debug", "d:t:O,client.trace");
```

```
/*
Tracing is available if you have compiled the driver using
cmake -DMYSQLCPPCONN_TRACE_ENABLE:BOOL=1
*/
con->setClientOption("client_trace", &on_off);
```

Chapter 5. MySQL Connector/C++ References

> **Caution**
>
> Please note, official support is not available for the beta version of MySQL Connector/C++.

See the JDBC overview for information on JDBC 4.0. Please also check the `examples/` directory of the download package.

Notes on using the MySQL Connector/C++ API

- `DatabaseMetaData::supportsBatchUpdates()` returns `true` because MySQL supports batch updates in general. However, no API calls for batch updates are provided by the MySQL Connector/C++ API.

- Two non-JDBC methods have been introduced for fetching and setting unsigned integers: `getUInt64()` and `getUInt()`. These are available for `ResultSet` and `Prepared_Statement`:

  - `ResultSet::getUInt64()`

  - `ResultSet::getUInt()`

  - `Prepared_Statement::setUInt64()`

  - `Prepared_Statement::setUInt()`

  The corresponding `getLong()` and `setLong()` methods have been removed.

- The method `DatabaseMetaData::getColumns()` has 23 columns in its result set, rather than the 22 columns defined by JDBC. The first 22 columns are as described in the JDBC documentation, but column 23 is new:

  23. `IS_AUTOINCREMENT`: String which is "YES" if the column is an auto-increment column. Otherwise the string contains "NO".

- MySQL Connector/C++ may return different metadata for the same column.

  When you have any column that accepts a charset and a collation in its specification and you specify a binary collation, such as:

  ```
  CHAR(250) CHARACTER SET 'latin1' COLLATE 'latin1_bin'
  ```

  The server sets the `BINARY` flag in the result set metadata of this column. The method `ResultSet-Metadata::getColumnTypeName()` uses the metadata and will report, due to the `BINARY` flag, that the column type name is `BINARY`. This is illustrated below:

  ```
  mysql> create table varbin(a varchar(20) character set utf8 collate utf8_bin);
  Query OK, 0 rows affected (0.00 sec)
  mysql> select * from varbin;
  Field   1:  `a`
  Catalog:    `def`
  Database:   `test`
  Table:      `varbin`
  Org_table:  `varbin`
  Type:       VAR_STRING
  Collation:  latin1_swedish_ci (8)
  Length:     20
  Max_length: 0
  Decimals:   0
  Flags:      BINARY
  0 rows in set (0.00 sec)
  mysql> select * from information_schema.columns where table_name='varbin'\G
  *************************** 1. row ***************************
             TABLE_CATALOG: NULL
              TABLE_SCHEMA: test
                TABLE_NAME: varbin
               COLUMN_NAME: a
          ORDINAL_POSITION: 1
            COLUMN_DEFAULT: NULL
               IS_NULLABLE: YES
                 DATA_TYPE: varchar
  CHARACTER_MAXIMUM_LENGTH: 20
    CHARACTER_OCTET_LENGTH: 60
         NUMERIC_PRECISION: NULL
             NUMERIC_SCALE: NULL
        CHARACTER_SET_NAME: utf8
            COLLATION_NAME: utf8_bin
               COLUMN_TYPE: varchar(20)
                COLUMN_KEY:
                     EXTRA:
                PRIVILEGES: select,insert,update,references
            COLUMN_COMMENT:
  1 row in set (0.01 sec)
  ```

However, `INFORMATION_SCHEMA` gives no hint in its `COLUMNS` table that metadata will contain the `BINARY` flag. `DatabaseMetaData::getColumns()` uses `INFORMATION_SCHEMA`. It will report the type name `CHAR` for the same column. Note, a different type code is also returned.

- The MySQL Connector/C++ class `sql::DataType` defines the following JDBC standard data types: `UNKNOWN`, `BIT`, `TINYINT`, `SMALLINT`, `MEDIUMINT`, `INTEGER`, `BIGINT`, `REAL`, `DOUBLE`, `DECIMAL`, `NUMERIC`, `CHAR`, `BINARY`, `VARCHAR`, `VARBINARY`, `LONGVARCHAR`, `LONGVARBINARY`, `TIMESTAMP`, `DATE`, `TIME`, `GEOMETRY`, `ENUM`, `SET`, `SQL-NULL`.

  However, the following JDBC standard data types are *not* supported by MySQL Connector/C++: `ARRAY`, `BLOB`, `CLOB`, `DISTINCT`, `FLOAT`, `OTHER`, `REF`, `STRUCT`.

- When inserting or updating `BLOB` or `TEXT` columns, MySQL Connector/C++ developers are advised not to use `setString()`. Instead it is recommended that the dedicated API function `setBlob()` be used instead.

  The use of `setString()` can cause a Packet too large error message. The error will occur if the length of the string passed to the connector using `setString()` exceeds `max_allowed_packet` (minus a few bytes reserved in the protocol for control purposes). This situation is not handled in MySQL Connector/C++, as this could lead to security issues, such as extremely large memory allocation requests due to malevolently long strings.

  However, if `setBlob()` is used, this problem does not arise. This is because `setBlob()` takes a streaming approach based on `std::istream`. When sending the data from the stream to MySQL Server, MySQL Connector/C++ will split the stream into chunks appropriate for MySQL Server and observe the `max_allowed_packet` setting currently being used.

  > **Caution**
  >
  > When using `setString()` it is not possible to set `max_packet_size` to a value large enough for the string, prior to passing it to MySQL Connector/C++. The MySQL 6.0 documentation for `max_packet_size` states: "As of MySQL 6.0.9, the session value of this variable is read only. Before 6.0.9, setting the session value is allowed but has no effect."

  This difference with the JDBC specification ensures that MySQL Connector/C++ is not vulnerable to memory flooding attacks.

- In general MySQL Connector/C++ works with MySQL 5.0, but it is not completely supported. Some methods may not be available when connecting to MySQL 5.0. This is because the Information Schema is used to obtain the requested information. There are no plans to improve the support for 5.0 because the current GA version of MySQL Server is 5.1. As a new product, MySQL Connector/C++ is primarily targeted at the MySQL Server GA version that was available on its release.

  The following methods will throw a `sql::MethodNotImplemented` exception when you connect to MySQL earlier than 5.1.0:

  - `DatabaseMetadata::getCrossReference()`

  - `DatabaseMetadata::getExportedKeys()`

- MySQL Connector/C++ includes a method `Connection::getClientOption()` which is not included in the JDBC API specification. The prototype is:

```
void getClientOption(const std::string & optionName, void * optionValue)
```

  The method can be used to check the value of connection properties set when establishing a database connection. The values are returned through the `optionValue` argument passed to the method with the type `void *`.

  Currently, `getClientOption()` supports fetching the `optionValue` of the following options:

  - `metadataUseInfoSchema`

  - `defaultStatementResultType`

  - `defaultPreparedStatementResultType`

  The connection option `metadataUseInfoSchema` controls whether to use the `Information_Schemata` for returning the meta data of `SHOW` commands. In the case of `metadataUseInfoSchema` the `optionValue` argument should be interpreted as a boolean upon return.

  In the case of both `defaultStatementResultType` and `defaultPreparedStatementResultType`, the `optionValue` argument should be interpreted as an integer upon return.

  The connection property can be either set when establishing the connection through the connection property map or using

`void Connection::setClientOption(const std::string & optionName, const void * option-Value)` where `optionName` is assigned the value `metadataUseInfoSchema`.

Some examples are given below:

```
int defaultStmtResType;
int defaultPStmtResType;
conn->getClientOption("defaultStatementResultType", (void *) &defaultStmtResType);
conn->getClientOption("defaultPreparedStatementResultType", (void *) &defaultPStmtResType);
bool isInfoSchemaUsed;
conn->getClientOption("metadataUseInfoSchema", (void *) &isInfoSchemaUsed);
```

- MySQL Connector/C++ also supports the following methods not found in the JDBC API standard:

```
std::string MySQL_Connection::getSessionVariable(const std::string & varname)
```

```
void MySQL_Connection::setSessionVariable(const std::string & varname, const std::string & value)
```

Note that both methods are members of the `MySQL_Connection` class. The methods get and set MySQL session variables.

`setSessionVariable()` is equivalent to executing:

```
SET SESSION <varname> = <value>
```

`getSessionVariable()` is equivalent to executing the following and fetching the first return value:

```
SHOW SESSION VARIABLES LIKE "<varname>"
```

You can use "%" and other placeholders in <varname>, if the underlying MySQL server supports this.

Chapter 6. MySQL Connector/C++ Known Bugs and Issues

> **Caution**
>
> Please note, official support is not available for the beta version of MySQL Connector/C++.

> **Note**
>
> Please report bugs through MySQL Bug System .

Known bugs:

None.

Known issues:

- When linking against a static library for 1.0.3 on Windows you need to define `CPPDBC_PUBLIC_FUNC` either in the compiler options (preferable) or with `/D "CPPCONN_PUBLIC_FUNC="`. You can also explicitly define it in your code by placing `#define CPPCONN_PUBLIC_FUNC` before the header inclusions.

- Generally speaking C++ library binaries are less portable than C library binaries. Issues can be caused by name mangling, different Standard Template Library (STL) versions and using different compilers and linkers for linking against the libraries than were used for building the library itself.

  Even a small change in the compiler version can, but does not have to, cause problems. If you obtain error messages, that you suspect are related to binary incompatibilities, build MySQL Connector/C++ from source, using the same compiler and linker that you will use to build and link your application.

  Due to the variations between Linux distributions, compiler and linker versions and STL versions, it is not possible to provide binaries for each and every possible configuration. However, the MySQL Connector/C++ binary distributions contain a `README` file that describes the environment and settings used to build the binary versions of the libraries.

- To avoid potential crashes the build configuration of MySQL Connector/C++ should match the build configuration of the application using it. For example, do not use the release build of MySQL Connector/C++ with a debug build of the client application.

See also the MySQL Connector/C++ Changelogs which can be found here Appendix A, *MySQL Connector/C++ Change History*.

Chapter 7. MySQL Connector/C++ Feature requests

> **Caution**
>
> Please note, official support is not available for the beta version of MySQL Connector/C++.

You can suggest new features in the first instance by joining the mailing list or forum and talking with the developers directly. See
MySQL Connector/C++ Support

The following feature requests are currently being worked on:

- C++ references for `Statements`, `ResultSets`, and exceptions, are being considered, instead of pointers to heap memory. This reduces the exception handling burden for the programmer.

- Adopt STL (suggestions are welcome).

- JDBC compliance: datatype interfaces and support through `ResultSet:getType()` and `PreparedState-ment:bind()`. Introduce `sql::Blob`, `sql::Clob`, `sql::Date`, `sql::Time`, `sql::Timestamp`, `sql::URL`. Support `get|setBlob()`, `get|setClob()`, `get|setDate()`, `get|setTime()`, `get|setTimestamp()`, `get|setURL()`

- Add support for all C-API connection options. Improved support for `mysql_options`.

- Add connect method which supports passing options using HashMaps.

- Create Windows installer.

Appendix A. MySQL Connector/C++ Change History

A.1. Changes in MySQL Connector/C++ 1.0.x

A.1.1. Changes in MySQL Connector/CPP 1.0.5 (21 April 2009)

This is the first Generally Available (GA) release.

Functionality added or changed:

- The interface of `sql::ConnectionMetaData`, `sql::ResultSetMetaData` and `sql::ParameterMetaData` was modified to have a protected destructor. As a result the client code has no need to destruct the metadata objects returned by the connector. MySQL Connector/C++ handles the required destruction. This enables statements such as:

  ```
  connection->getMetaData->getSchema();
  ```

  This avoids potential memory leaks that could occur as a result of losing the pointer returned by `getMetaData()`.

- Improved memory management. Potential memory leak situations are handled more robustly.

- Changed the interface of `sql::Driver` and `sql::Connection` so they accept the options map by alias instead of by value.

- Changed the return type of `sql::SQLException::getSQLState()` from `std::string` to `const char *` to be consistent with `std::exception::what()`.

- Implemented `getResultSetType()` and `setResultSetType()` for `Statement`. Uses `TYPE_FORWARD_ONLY`, which means unbuffered result set and `TYPE_SCROLL_INSENSITIVE`, which means buffered result set.

- Implemented `getResultSetType()` for `PreparedStatement`. The setter is not implemented because currently `PreparedStatement` cannot do refetching. Storing the result means the bind buffers will be correct.

- Added the option `defaultStatementResultType` to `MySQL_Connection::setClientOption()`. Also, the method now returns `sql::Connection *`.

- Added `Result::getType()`. Implemented for the three result set classes.

- Enabled tracing functionality when building with Microsoft Visual C++ 8 and later, which corresponds to Microsoft Visual Studio 2005 and later.

- Added better support for named pipes, on Windows. Use `pipe://` and add the path to the pipe. Shared memory connections are currently not supported.

Bugs fixed:

- A bug was fixed in `MySQL_Connection::setSessionVariable()`, which had been causing exceptions to be thrown.

A.1.2. Changes in MySQL Connector/CPP 1.0.4 (31 March 2009 beta)

Functionality added or changed:

- An installer was added for the Windows operating system.

- Minimum CMake version required was changed from 2.4.2 to 2.6.2. The latest version is required for building on Windows.

- `metadataUseInfoSchema` was added to the connection property map, which allows control of the `INFORMATION_SCHEMA` for meta data.

- Implemented `MySQL_ConnectionMetaData::supportsConvert(from, to)`.

- Added support for MySQL Connector/C.

Bugs fixed:

- A bug was fixed in all implementations of `ResultSet::relative()` which was giving a wrong return value although positioning was working correctly.

- A leak was fixed in `MySQL_PreparedResultSet`, which occurred when the result contained a `BLOB` column.

A.1.3. Changes in MySQL Connector/CPP 1.0.3 (02 March 2009 alpha)

Functionality added or changed:

- Added new tests in `test/unit/classes`. Those tests are mostly about code coverage. Most of the actual functionality of the driver is tested by the tests found in `test/CJUnitPort`.

- New data types added to the list returned by `DatabaseMetaData::getTypeInfo()` are `FLOAT UNSIGED`, `DECIMAL UNSIGNED`, `DOUBLE UNSIGNED`. Those tests may not be in the JDBC specification. However, due to the change you should be able to look up every type and type name returned by, for example, `ResultSetMetaData::getColumnTypeName()`.

- `MySQL_Driver::getPatchVersion` introduced.

- Major performance improvements due to new buffered `ResultSet` implementation.

- Addition of `test/unit/README` with instructions for writing bug and regression tests.

- Experimental support for STLPort. This feature may be removed again at any time later without prior warning! Type `cmake -L` for configuration instructions.

- Added properties enabled methods for connecting, which add many connect options. This uses a dictionary (map) of key value pairs. Methods added are `Driver::connect(map)`, and `Connection::Connection(map)`.

- New BLOB implementation. `sql::Blob` was removed in favor of `std::istream`. C++'s `IOStream` library is very powerful, similar to PHP's streams. It makes no sense to reinvent the wheel. For example, you can pass a `std::istringstream` object to `setBlob()` if the data is in memory, or just open a file `std::fstream` and let it stream to the DB, or write its own stream. This is also true for `getBlob()` where you can just copy data (if a buffered result set), or stream data (if implemented).

- Implemented `ResultSet::getBlob()` which returns `std::stream`.

- Fixed `MySQL_DatabaseMetaData::getTablePrivileges()`. Test cases were added in the first unit testing framework.

- Implemented `MySQL_Connection::setSessionVariable()` for setting variables like `sql_mode`.

- Implemented `MySQL_DatabaseMetaData::getColumnPrivileges()`.

- `cppconn/datatype.h` has changed and is now used again. Reimplemented the type subsystem to be more usable - more types for binary and nonbinary strings.

- Implementation for `MySQL_DatabaseMetaData::getImportedKeys()` for MySQL versions before 5.1.16 using `SHOW`, and above using `INFORMATION_SCHEMA`.

- Implemented `MySQL_ConnectionMetaData::getProcedureColumns()`.

- `make package_source` now packs with bzip2.

- Re-added `getTypeInfo()` with information about all types supported by MySQL and the `sql::DataType`.

- Changed the implementation of `MySQL_ConstructedResultSet` to use the more efficient O(1) access method. This should improve the speed with which the metadata result sets are used. Also, there is less copying during the construction of the result set, which means that all result sets returned from the meta data functions will be faster.

- Introduced, internally, `sql::mysql::MyVal` which has implicit constructors. Used in `mysql_metadata.cpp` to create result sets with native data instead of always string (varchar).

- Renamed `ResultSet::getLong()` to `ResultSet::getInt64()`. `resultset.h` includes typdefs for Windows to be able to use `int64_t`.

- Introduced `ResultSet::getUInt()` and `ResultSet::getUInt64()`.

- Improved the implementation for `ResultSetMetaData::isReadOnly()`. Values generated from views are read-only. These generated values don't have `db` in `MYSQL_FIELD` set, while all normal columns do have.

- Implemented `MySQL_DatabaseMetaData::getExportedKeys()`.

- Implemented `MySQL_DatabaseMetaData::getCrossReference()`.

Bugs fixed:

- Bug fixed in `MySQL_PreparedResultSet::getString()`. Returned string that had real data but the length was random. Now, the string is initialized with the correct length and thus is binary safe.

- Corrected handling of unsigned server types. Now returning correct values.

- Fixed handling of numeric columns in `ResultSetMetaData::isCaseSensitive` to return `false`.

A.1.4. Changes in MySQL Connector/CPP 1.0.2 (19 December 2008 alpha)

Functionality added or changed:

- Implemented `getScale()`, `getPrecision()` and `getColumnDisplaySize()` for `MySQL_ResultSetMetaData` and `MySQL_Prepared_ResultSetMetaData`.

- Changed `ResultSetMetaData` methods `getColumnDisplaySize()`, `getPrecision()`, `getScale()` to return `unsigned int` instead of `signed int`.

- `DATE`, `DATETIME` and `TIME` are now being handled when calling the `MySQL_PreparedResultSet` methods `getString()`, `getDouble()`, `getInt()`, `getLong()`, `getBoolean()`.

- Reverted implementation of `MySQL_DatabaseMetaData::getTypeInfo()`. Now unimplemented. In addition, removed `cppconn/datatype.h` for now, until a more robust implementation of the types can be developed.

- Implemented `MySQL_PreparedStatement::setNull()`.

- Implemented `MySQL_PreparedStatement::clearParameters()`.

- Added PHP script `examples/cpp_trace_analyzer.php` to filter the output of the debug trace. Please see the inline comments for documentation. This script is unsupported.

- Implemented `MySQL_ResultSetMetaData::getPrecision()` and `MySQL_Prepared_ResultSetMetaData::getPrecision()`, updating example.

- Added new unit test framework for JDBC compliance and regression testing.

- Added `test/unit` as a basis for general unit tests using the new test framework, see `test/unit/example` for basic usage examples.

Bugs fixed:

- Fixed `MySQL_PreparedStatementResultSet::getDouble()` to return the correct value when the underlying type is `MYSQL_TYPE_FLOAT`.

- Fixed bug in `MySQL_ConnectionMetaData::getIndexInfo()`. The method did not work because the schema name wasn't included in the query sent to the server.

- Fixed a bug in `MySQL_ConnectionMetaData::getColumns()` which was performing a cartesian product of the columns in the table times the columns matching `columnNamePattern`. The example `example/connection_meta_schemaobj.cpp` was extended to cover the function.

- Fixed bugs in `MySQL_DatabaseMetaData`. All `supportsCatalogXXXXX` methods were incorrectly returning `true` and all `supportsSchemaXXXX` methods were incorrectly returning `false`. Now `supportsCatalogXXXXX` returns `false` and `supportsSchemaXXXX` returns `true`.

- Fixed bugs in the `MySQL_PreparedStatements` methods `setBigInt()` and `setDatetime()`. They decremented the internal column index before forwarding the request. This resulted in a double-decrement and therefore the wrong internal column index. The error message generated was:

```
setString() ... invalid "parameterIndex"
```

- Fixed a bug in `getString()`. `getString()` is now binary safe. A new example was also added.

- Fixed bug in `FLOAT` handling.

- Fixed `MySQL_PreparedStatement::setBlob()`. In the tests there is a simple example of a class implementing `sql::Blob`.

A.1.5. Changes in MySQL Connector/CPP 1.0.1 (01 December 2008 alpha)

Functionality added or changed:

- `sql::mysql::MySQL_SQLException` was removed. The distinction between server and client (connector) errors, based on the type of the exception, has been removed. However, the error code can still be checked in order to evaluate the error type.

- Support for (n)make install was added. You can change the default installation path. Carefully read the messages displayed after executing cmake. The following are installed:

  - Static and the dynamic version of the library, `libmysqlcppconn`.

  - Generic interface, `cppconn`.

  - Two MySQL specific headers:

    `mysql_driver.h`, use this if you want to get your connections from the driver instead of instantiating a `MySQL_Connection` object. This makes your code portable when using the common interface.

    `mysql_connection.h`, use this if you intend to link directly to the `MySQL_Connection` class and use its specifics not found in `sql::Connection`.

    However, you can make your application fully abstract by using the generic interface rather than these two headers.

- Driver Manager was removed.

- Added `ConnectionMetaData::getSchemas()` and `Connection::setSchema()`.

- `ConnectionMetaData::getCatalogTerm()` returns not applicable, there is no counterpart to catalog in MySQL Connector/C++.

- Added experimental GCov support, `cmake -DMYSQLCPPCONN_GCOV_ENABLE:BOOL=1`

- All examples can be given optional connection parameters on the command line, for example:

```
examples/connect tcp://host:port user pass database
```

or

```
examples/connect unix:///path/to/mysql.sock user pass database
```

- Renamed `ConnectionMetaData::getTables: TABLE_COMMENT` to `REMARKS`.

- Renamed `ConnectionMetaData::getProcedures: PROCEDURE_SCHEMA` to `PROCEDURE_SCHEM`.

- Renamed `ConnectionMetaData::getPrimaryKeys(): COLUMN` to `COLUMN_NAME`, `SEQUENCE` to `KEY_SEQ`, and `INDEX_NAME` to `PK_NAME`.

- Renamed `ConnectionMetaData::getImportedKeys(): PKTABLE_CATALOG` to `PKTABLE_CAT`, `PKTABLE_SCHEMA` to `PKTABLE_SCHEM`, `FKTABLE_CATALOG` to `FKTABLE_CAT`, `FKTABLE_SCHEMA` to `FKTABLE_SCHEM`.

- Changed metadata column name `TABLE_CATALOG` to `TABLE_CAT` and `TABLE_SCHEMA` to `TABLE_SCHEM` to ensure JDBC compliance.

- Introduced experimental CPack support, see make help.

- All tests changed to create TAP compliant output.

- Renamed `sql::DbcMethodNotImplemented` to `sql::MethodNotImplementedException`

- Renamed `sql::DbcInvalidArgument` to `sql::InvalidArgumentException`

- Changed `sql::DbcException` to implement the interface of JDBC's `SQLException`. Renamed to `sql::SQLException`.

- Converted Connector/J tests added.

- MySQL Workbench 5.1 changed to use MySQL Connector/C++ for its database connectivity.

- New directory layout.