MySQL Connector/C++ 1.1 Developer Guide

Abstract

This manual describes how to install and configure MySQL Connector/C++ 1.1, which provides a C++ interface for communicating with MySQL servers, and how to use Connector/C++ to develop database applications.

Connector/C++ 8.0 is highly recommended for use with MySQL Server 8.0 and 5.7. Please upgrade from Connector/C++ 1.1 to Connector/C++ 8.0.

For notes detailing the changes in each release of Connector/C++, see MySQL Connector/C++ Release Notes.

For legal information, see the Legal Notices.

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

Licensing information. This product may include third-party software, used under license. If you are using a *Commercial* release of MySQL Connector/C++, see the MySQL Connector/C++ Commercial Release License Information User Manual for licensing information, including licensing information relating to third-party software that may be included in this Commercial release. If you are using a *Community* release of MySQL Connector/C++, see the MySQL Connector/C++ Community Release License Information User Manual for licensing information, including licensing information relating to third-party software that may be included in this Community release.

Document generated on: 2019-04-09 (revision: 61594)

Table of Contents

Preface and Legal Notices

This manual describes how to install and configure MySQL Connector/C++ 1.1, and how to use it to develop database applications.

Legal Notices

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Chapter 1 Introduction to Connector/C++

MySQL Connector/C++ 1.1 is a MySQL database connector for C++ applications that connect to MySQL servers. Connector/C++ enables development of C++ applications that use the JDBC-based API.

For more detailed requirements about required MySQL versions for Connector/C++ applications, see Platform Support and Prerequisites.

For notes detailing the changes in each release of Connector/C++, see MySQL Connector/C++ Release Notes.

Connector/C++ Benefits

MySQL Connector/C++ offers the following benefits for C++ users compared to the MySQL C API provided by the MySQL client library:

- Convenience of pure C++.
- Supports a JDBC 4.0-based API.
- Supports the object-oriented programming paradigm.
- Reduces development time.
- Licensed under the GPL with the FLOSS License Exception.
- Available under a commercial license upon request.

Connector/C++ and JDBC Compatibility

Connector/C++ is compatible with the JDBC 4.0 API. Connector/C++ does not implement the entire JDBC 4.0 API, but does feature these classes: `Connection`, `DatabaseMetaData`, `Driver`, `PreparedStatement`, `ResultSet`, `ResultSetMetaData`, `Savepoint`, `Statement`.

The JDBC 4.0 API defines approximately 450 methods for the classes just mentioned. Connector/C++ implements approximately 80% of these.

Platform Support and Prerequisites

To see which platforms are supported, visit the Connector/C++ downloads page.

For Connector/C++ 1.1.11 and higher, Commercial and Community distributions require the Visual C++ Redistributable for Visual Studio 2015 to work on Windows platforms. Up through Connector/C++ 1.1.10, Community (not Commercial) distributions require the Visual C++ Redistributable for Visual Studio 2013. The Redistributable is available at the Microsoft Download Center; install it before installing Connector/C++.

These requirements apply to building and running Connector/C++ applications, and to building Connector/C++ itself if you build it from source:

- To build Connector/C++ applications:
  - The MySQL version does not apply.
  - On Windows, Microsoft Visual Studio 2015 is required.
- To run Connector/C++ applications, a server from MySQL 5.5 or higher is required.
- To build Connector/C++ from source:
  - Building Connector/C++ requires the client library from MySQL 5.7 (5.7.9 or higher) or MySQL 8.0 (8.0.11 or higher).
  - On Windows, Microsoft Visual Studio 2015 is required.

Chapter 2 Obtaining Connector/C++

Connector/C++ binary and source distributions are available, in platform-specific packaging formats. To obtain a distribution, visit the Connector/C++ downloads page. It is also possible to clone the Connector/C++ Git source repository.

- Connector/C++ binary distributions are available for Microsoft Windows, and for Unix and Unix-like platforms. See Chapter 3, *Installing Connector/C++ from a Binary Distribution*.
- Connector/C++ source distributions are available as compressed `tar` files or Zip archives and can be used on any supported platform. See Chapter 4, *Installing Connector/C++ from Source*.
- The Connector/C++ source code repository uses Git and is available at GitHub. See Chapter 4, *Installing Connector/C++ from Source*.

Chapter 3 Installing Connector/C++ from a Binary Distribution

To obtain a Connector/C++ binary distribution, visit the Connector/C++ downloads page.

Installation on Windows

> **Important**
>
> For Connector/C++ 1.1.11 and higher, Commercial and Community distributions require the Visual C++ Redistributable for Visual Studio 2015 to work on Windows platforms. Up through Connector/C++ 1.1.10, Community (not Commercial) distributions require the Visual C++ Redistributable for Visual Studio 2013.
>
> The Redistributable is available at the Microsoft Download Center; install it before installing any version of Connector/C++ that requires it.

These binary-distribution installation methods are available on Windows:

- **MySQL Installer.** The simplest and recommended method for installing Connector/C++ on Windows platforms is to download *MySQL Installer* and let it install and configure all the MySQL products on your system. For details, see MySQL Installer for Windows.
- **Windows MSI installer.** To use the MSI Installer (`.msi` file), launch it and follow the prompts in the screens it presents to install Connector/C++ in the location of your choosing.

  The MSI Installer requires administrative privileges. It begins by presenting a welcome screen that enables you to continue the installation or cancel it. If you continue the installation, the MSI Installer overview screen enables you to select the type of installation to perform:

  - The **Typical** installation consists of all required header files and the Release libraries.
  - The **Custom** installation enables you to install additional Debug versions of the connector libraries. If you select this installation type, the MSI Installer presents a Custom Setup screen that enables you to select which features to install and where to install them.
  - The **Complete** installation installs everything in the distribution.
- **Zip archive package without installer.** To install from a Zip archive package (`.zip` file), use `WinZip` or another tool that can read `.zip` files to unpack the file into the location of your choosing.

Post-Installation Setup

If you plan to use a dynamically linked version of Connector/C++, the `libmysqlclient` MySQL client library must be registered with the dynamic linker so that it can be found at runtime. Make sure that your system can reference the MySQL client library that Connector/C++ is linked against and thus requires. Consult your operating system documentation on how to modify and expand the search path for libraries. Many Unix and Unix-like systems enable configuring dynamic library locations using `LD_LIBRARY_PATH` environment variable. For example, if you install Connector/C++ under `/usr/local/lib`, try this:

```
export LD_LIBRARY_PATH=/usr/local/lib
```

On macOS, try this:

```
export DYLD_LIBRARY_PATH=/usr/local/lib
```

It may also be necessary to run `ldconfig` or equivalent utility.

If you cannot modify the library search path, it may help to copy your application, the Connector/C++ library and the MySQL client library into the same directory. Most systems search for libraries in the current directory.

Chapter 4 Installing Connector/C++ from Source

Table of Contents

This chapter describes how to install Connector/C++ using a source distribution or a copy of the Git source repository.

4.1 Source Installation System Prerequisites

To install Connector/C++ from source, the following system requirements must be satisfied:

- Build Tools
- MySQL Client Library
- Boost C++ Libraries

Build Tools

You must have the cross-platform build tool `CMake` (2.8.12 or higher is recommended; older versions may work).

You must have a C++ compiler that supports C++11.

MySQL Client Library

Building Connector/C++ from source requires the client library from MySQL 5.7 (5.7.9 or higher) or MySQL 8.0 (8.0.11 or higher).

Typically, the MySQL client library is installed when MySQL is installed. However, check your operating system documentation for other installation options.

To specify where to find the client library, set the `MYSQL_DIR` `CMake` option appropriately at configuration time as necessary (see Section 4.6, “Connector/C++ Source-Configuration Options”).

Boost C++ Libraries

To compile Connector/C++, the Boost C++ libraries are always needed. Boost 1.59.0 or newer must be installed. To obtain Boost and its installation instructions, visit the official Boost site.

After Boost is installed, use the `BOOST_ROOT` `CMake` option to indicate where the Boost files are located (see Section 4.6, “Connector/C++ Source-Configuration Options”):

```
cmake [other_options] -DBOOST_ROOT=/usr/local/boost_1_59_0
```

Adjust the path as necessary to match your installation.

4.2 Obtaining and Unpacking a Connector/C++ Source Distribution

To obtain a Connector/C++ source distribution, visit the Connector/C++ downloads page. Alternatively, clone the Connector/C++ Git source repository.

A Connector/C++ source distribution is packaged as a compressed `tar` file or Zip archive, denoted here as *`PACKAGE`*`.tar.gz` or *`PACKAGE`*`.zip`. A source distribution in `tar` file or Zip archive format can be used on any supported platform.

To unpack a compressed `tar` file, use this command in the intended installation directory:

```
tar zxvf PACKAGE.tar.gz
```

After unpacking the distribution, build it using the appropriate instructions for your platform later in this chapter.

To install from a Zip archive package (`.zip` file), use `WinZip` or another tool that can read `.zip` files to unpack the file into the location of your choosing. After unpacking the distribution, build it using the appropriate instructions for your platform later in this chapter.

To clone the Connector/C++ code from the source code repository located on GitHub at https://github.com/mysql/mysql-connector-cpp, use this command:

```
git clone https://github.com/mysql/mysql-connector-cpp.git
```

That command should create a `mysql-connector-cpp` directory containing a copy of the entire Connector/C++ source tree.

The `git clone` command sets the sources to the `master` branch, which is the branch that contains the latest sources. Released code is in the `8.0` and `1.1` branches (the `8.0` branch contains the same sources as the `master` branch). If necessary, use `git checkout` in the source directory to select the desired branch. For example, to build Connector/C++ 8.0:

```
cd mysql-connector-cpp
git checkout 8.0
```

Or to build Connector/C++ 1.1:

```
cd mysql-connector-cpp
git checkout 1.1
```

After cloning the repository, build it using the appropriate instructions for your platform later in this chapter.

After the initial checkout operation to get the source tree, run `git pull` periodically to update your source to the latest version.

4.3 Installing Connector/C++ from Source on Unix and Unix-Like Systems

To install Connector/C++ from source, verify that your system satisfies the requirements outlined in Section 4.1, “Source Installation System Prerequisites”.

1. Change location to the top-level directory of your Connector/C++ source distribution, then run `CMake` to build a `Makefile`:

```
cmake .
```

To use configuration values different from the defaults, use the options described at Section 4.6, “Connector/C++ Source-Configuration Options”. For example, to specify the installation location explicitly, use the `CMAKE_INSTALL_PREFIX` option:

```
-DCMAKE_INSTALL_PREFIX=path_name
```

`CMake` checks to see whether the `MYSQL_CONFIG_EXECUTABLE` `CMake` option is set. If not, `CMake` tries to locate `mysql_config` in the default locations.

2. Use `make` to build Connector/C++. First make sure you have a clean build, then build the connector:

```
make clean
make
```

If all goes well, you will find the Connector/C++ library in the `driver` directory.

3. Install the header and library files:

```
make install
```

Unless you have changed the installation location in the configuration step by specifying the `CMAKE_INSTALL_PREFIX` `CMake` option, `make install` copies the header files to the directory `/usr/local/include`. The `mysql_connection.h` and `mysql_driver.h` header files are copied.

Again, unless you have specified otherwise, `make install` copies the library files to `/usr/local/lib`. The files copied are the dynamic library `libmysqlcppconn.so`, and the static library `libmysqlcppconn-static.a`. The dynamic library file name extension might differ on your system (for example, `.dylib` on macOS).

After installing Connector/C++, you can carry out a quick test to check the installation. To do this, compile and run one of the example programs, such as `examples/standalone_example.cpp`. This example is discussed in more detail later, but for now, you can use it to test whether the connector has been correctly installed. This procedure assumes availability of a working MySQL server to which you can connect. It also assumes header and library locations of `/usr/local/include` and `/usr/local/lib`, respectively; adjust these as necessary for your system.

1. Compile the example program. To do this, change location to the `examples` directory and enter this command:

```
g++ -o test_install \
  -I/usr/local/include -I/usr/local/include/cppconn \
  -Wl,-Bdynamic standalone_example.cpp -lmysqlcppconn
```

2. Make sure the dynamic library which is used in this case can be found at runtime:

```
export LD_LIBRARY_PATH=/usr/local/lib
```

On macOS, try this:

```
export DYLD_LIBRARY_PATH=/usr/local/lib
```

It may also be necessary to run `ldconfig` or equivalent utility.

3. Now run the program to test your installation, substituting the appropriate host, user, password, and database arguments for your system:

```
./test_install localhost root password database
```

You should see output similar to the following:

```
Connector/C++ standalone program example...

... running 'SELECT 'Welcome to Connector/C++' AS _message'
... MySQL replies: Welcome to Connector/C++
... say it again, MySQL
....MySQL replies: Welcome to Connector/C++

... find more at http://www.mysql.com
```

4.4 Installing Connector/C++ from Source on Windows

To install Connector/C++ from source, verify that your system satisfies the requirements outlined in Section 4.1, “Source Installation System Prerequisites”.

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

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

```
cmake --help
```

It is likely that your version of `CMake` supports more compilers, known by `CMake` as *generators*, than can actually be used to build Connector/C++. We have built Connector/C++ using the following generators:

- Microsoft Visual Studio 2013
- NMake

`CMake` makes it easy for you to try other compilers. However, you may experience compilation warnings or errors, or linking issues not detected by Visual Studio.

Use these steps to build Connector/C++:

1. Change location to the top-level directory of your Connector/C++ source distribution, then run `CMake` to generate build files for your generator.

   For Visual Studio:

```
cmake -G "Visual Studio 12 2013"
```

   For NMake:

```
```

```
cmake -G "NMake Makefiles"
```

To use configuration values different from the defaults, use the options described at Section 4.6, “Connector/C++ Source-Configuration Options”.

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

   For Visual Studio:

   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, debug and nondebug versions among them.

   For example, to use the command line, execute commands like these to build the distribution and create a Zip package:

```
devenv MYSQLCPPCONN.sln /build RelWithDebInfo
cpack -G ZIP --config CPackConfig.cmake -C RelWithDebInfo
```

   For NMake:

```
nmake
```

4.5 Dynamically Linking Connector/C++ Against the MySQL Client Library

> **Note**
>
> This section refers to dynamic linking of Connector/C++ to the client library, not dynamic linking of applications to Connector/C++. Precompiled binaries of Connector/C++ use static binding with the client library by default.

An application that uses Connector/C++ can be linked either statically or dynamically to the Connector/C++ libraries. Connector/C++ is usually linked statically to the underlying MySQL client library.

It is also possible to link dynamically to the underlying MySQL client library, although this capability is not enabled by default. To link Connector/C++ dynamically to the client library, disable `MYSQLCLIENT_STATIC_LINKING` when building Connector/C++ from source:

```
rm CMakeCache.txt
cmake . -DMYSQLCLIENT_STATIC_LINKING=0
make clean
make
make install
```

Now, when creating a connection in your application, Connector/C++ selects and loads a client library at runtime. It chooses the client library by searching defined locations and environment variables depending on the host operating system. It is also possible when creating a connection in an application to define an absolute path to the client library to load at runtime. This can be convenient if you have defined a standard location from which you want the client library to be loaded. This is sometimes done to circumvent possible conflicts with other versions of the client library that may be located on the system.

If `MYSQLCLIENT_STATIC_LINKING` is disabled to enable dynamic linking to the MySQL client library, `MYSQLCLIENT_STATIC_BINDING` determines whether to link to the shared MySQL client library. By

default, MYSQLCLIENT_STATIC_BINDING is enabled. If MYSQLCLIENT_STATIC_BINDING is disabled, Connector/C++ is not linked to the shared MySQL client library. Instead, that library is loaded and mapped at runtime.

4.6 Connector/C++ Source-Configuration Options

Connector/C++ recognizes the CMake options described in this section.

Table 4.1 Connector/C++ Source-Configuration Option Reference

Formats	Description	Default	Introduced
BOOST_ROOT	The Boost source directory		
BUNDLE_DEPENDENCIES	Whether to bundle external dependency libraries with the connector	OFF	1.1.11
CMAKE_BUILD_TYPE	Type of build to produce	Debug	
CMAKE_ENABLE_C++11	Whether to enable C++11 support	OFF	1.1.6
CMAKE_INSTALL_PREFIX	Installation base directory	/usr/local	
MYSQLCLIENT_NO_THREADS	Whether to link against single-threaded MySQL client library	OFF	
MYSQLCLIENT_STATIC_BINDING	Whether to link to the shared MySQL client library	ON	
MYSQLCLIENT_STATIC_LINKING	Whether to statically link to the MySQL client library	ON	
MYSQLCPPCONN_GCOV_ENABLE	Whether to enable gcov support	OFF	
MYSQLCPPCONN_TRACE_ENABLE	Whether to enable tracing functionality	OFF	
MYSQL_CFLAGS	C compiler flags		
MYSQL_CONFIG_EXECUTABLE	Path to the mysql_config program	${MYSQL_DIR}/bin/mysql_config	
MYSQL_CXXFLAGS	C++ compiler flags		
MYSQL_CXX_LINKAGE	Whether MySQL client library needs C++ linking	ON	
MYSQL_DIR	MySQL Server or Connector/C installation directory		
MYSQL_EXTRA_LIBRARIES	Extra link libraries		
MYSQL_INCLUDE_DIR	The MySQL header file directory	${MYSQL_DIR}/include	
MYSQL_LIB_DIR	The MySQL client library directory	${MYSQL_DIR}/lib	
MYSQL_LINK_FLAGS	Extra link flags		
USE_SERVER_CXXFLAGS	Use MySQL Server CXXFLAGS value rather than system default	OFF	1.1.7

- -DBOOST_ROOT=*dir_name*

  The directory where the Boost sources are installed.

- `-DBUNDLE_DEPENDENCIES=bool`

  This is an internal option used for creating Connector/C++ distribution packages.

- `-DCMAKE_BUILD_TYPE=type`

  The type of build to produce:

  - `Debug`: Disable optimizations and generate debugging information. This is the default.
  - `Release`: Enable optimizations.
  - `RelWithDebInfo`: Enable optimizations and generate debugging information.

- `-DCMAKE_ENABLE_C++11=bool`

  Whether to enable C++11 support. The default is `OFF`.

- `-DCMAKE_INSTALL_PREFIX=dir_name`

  The installation base directory (where to install Connector/C++).

- `-DMYSQLCLIENT_NO_THREADS=bool`

  Whether to link against a single-threaded `libmysqlclient` MySQL client library.

  This option is obsolete; `libmysqlclient` is always multithreaded.

- `-DMYSQLCLIENT_STATIC_BINDING=bool`

  Whether to link to the shared MySQL client library. This option is used only if `MYSQLCLIENT_STATIC_LINKING` is disabled. disabled to enable dynamic linking to the MySQL client library. In that case, if `MYSQLCLIENT_STATIC_BINDING` is enabled (the default), Connector/C++ is linked to the shared MySQL client library. Otherwise, the shared MySQL client library is loaded and mapped at runtime. For more information, see Section 4.5, “Dynamically Linking Connector/C++ Against the MySQL Client Library”.

- `-DMYSQLCLIENT_STATIC_LINKING=bool`

  Whether to link statically to the MySQL client library. The default is `ON` (use static linking to the client library). Disabling this option enables dynamic linking to the client library. For more information, see Section 4.5, “Dynamically Linking Connector/C++ Against the MySQL Client Library”.

- `-DMYSQLCPPCONN_GCOV_ENABLE=bool`

  Whether to enable `gcov` support.

- `-DMYSQLCPPCONN_TRACE_ENABLE=VALUE_TYPE`

  Whether to enable tracing functionality. For information about tracing, see Chapter 8, *Connector/C++ Debug Tracing*.

- `-DMYSQL_CFLAGS=flags`

  C compiler flags.

- `-DMYSQL_CONFIG_EXECUTABLE=file_name`

  The path to the `mysql_config` program.

On non-Windows systems, `CMake` checks to see whether `MYSQL_CONFIG_EXECUTABLE` is set. If not, `CMake` tries to locate `mysql_config` in the default locations.

- `-DMYSQL_CXXFLAGS=`*`flags`*

  C++ compiler flags.

- `-DMYSQL_CXX_LINKAGE=`*`bool`*

  Whether the MySQL client library needs C++ linking.

- `-DMYSQL_DIR=`*`dir_name`*

  The directory where MySQL is installed.

- `-DMYSQL_EXTRA_LIBRARIES=`*`flags`*

  Any required additional link libraries.

- `-DMYSQL_INCLUDE_DIR=`*`dir_name`*

  The directory where the MySQL header files are installed.

- `-DMYSQL_LIB_DIR=`*`dir_name`*

  The directory where the MySQL client library is installed.

- `-DMYSQL_LINK_FLAGS=`*`flags`*

  Any required additional link flags.

- `-DUSE_SERVER_CXXFLAGS=`*`bool`*

  Use MySQL `CXXFLAGS` values rather than the system default value.

Chapter 5 Building Connector/C++ Applications

Table of Contents

This chapter provides guidance on building Connector/C++ applications:

- Prerequisites that must be satisfied to build Connector/C++ applications successfully. See Section 5.1, “Building Connector/C++ Applications: General Considerations”.
- Building applications for Connector/C++ on Windows using Visual Studio. See Section 5.2, “Building Connector/C++ Applications on Windows with Microsoft Visual Studio”.
- Building applications for Connector/C++ on Linux using NetBeans. See Section 5.3, “Building Connector/C++ 1.1 Applications on Linux with NetBeans”.

For discussion of which programming interfaces are available in Connector/C++, see Chapter 1, *Introduction to Connector/C++*.

5.1 Building Connector/C++ Applications: General Considerations

Keep these general considerations in mind for building Connector/C++ applications:

- Build Tools and Configuration Settings
- Boost Header Files
- Runtime Libraries

Build Tools and Configuration Settings

It is important that the tools you use to build your Connector/C++ applications are compatible with the tools used to build Connector/C++ itself. Ideally, build your applications with the same tools that were used to build the Connector/C++ binaries.

To avoid issues, ensure that these factors are the same for your applications and Connector/C++ itself:

- Compiler version.
- Runtime library.
- Runtime linker configuration settings.

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

To use a different compiler version, release configuration, or runtime library, first build Connector/C++ from source using your desired settings (see Chapter 4, *Installing Connector/C++ from Source*), then build your applications using those same settings.

Connector/C++ binary distribution packages include a `BUILDINFO.txt` file that describes the environment and configuration options used to build the distribution. If you installed Connector/C++ from a binary

distribution and experience build-related issues on a platform, it may help to check the settings that were used to build the distribution on that platform.

Boost Header Files

To compile applications that use Connector/C++, the Boost header files are always needed. Boost 1.59.0 or newer must be installed, and the location of the headers must be added to the include path. To obtain Boost and its installation instructions, visit the official Boost site.

Runtime Libraries

If an application is built using dynamic link libraries, those libraries must be present not just on the build host, but on target hosts where the application runs. The libraries and their runtime dependencies must be found by the dynamic linker. The dynamic linker must be properly configured to find Connector/C++ libraries and their dependencies.

On Windows, target hosts running the client application must have the Visual C++ Redistributable for Visual Studio installed. The required version is VC++ Redistributable 2015.

5.2 Building Connector/C++ Applications on Windows with Microsoft Visual Studio

This section describes how to build Connector/C++ applications for Windows using Microsoft Visual Studio. For general application-building information, see Section 5.1, “Building Connector/C++ Applications: General Considerations”.

Developers using Microsoft Windows must satisfy the following requirements:

- Microsoft Visual Studio 2015 is required to build Connector/C++ applications.
- Your applications should use the same linker configuration as Connector/C++. For example, use one of `/MD`, `/MDd`, `/MT`, or `/MTd`.

Linking Connector/C++ to Applications

Connector/C++ is available as a dynamic or static library to use with your application. This section describes how to link the library to your application.

Linking the Dynamic Library

To use a dynamic library file (`.dll` extension), link your application with a `.lib` import library. At runtime, the application must have access to the `.dll` library.

The following table indicates which import library and dynamic library files to use for Connector/C++. *`LIB`* denotes the Connector/C++ installation library path name.

Table 5.1 Connector/C++ Import Library and Dynamic Library Names

Import Library	Dynamic Library
`LIB``/mysqlcppconn.lib`	*`LIB`*`/mysqlcppconn.dll`

Linking the Static Library

To use a static library file (`.lib` extension), link your application with the library.

The static library name is *`LIB`*`/mysqlcppconn-static.lib`, where *`LIB`* denotes the Connector/C++ installation library path name.

Building Connector/C++ Applications with Microsoft Visual Studio

The initial steps for building an application to use either the dynamic or static library are the same. Some additional steps vary, depend on whether you are building your application to use the dynamic or static library.

1. Start a new Visual C++ project in Visual Studio.
2. In the drop-down list for build configuration on the toolbar, change the configuration from the default option of **Debug** to **Release**.

   > **Connector/C++ and Application Build Configuration Must Match**
   >
   > Because the application build configuration must match that of the Connector/C++ it uses, **Release** is required when using an Oracle-built Connector/C++, which is built in the release configuration. When linking dynamically, it is possible to build your code in debug mode even if the connector libraries are built in release mode. However, in that case, it will not be possible to step inside connector code during a debug session. To be able to do that, or to build in debug mode while linking statically to the connector, you must build Connector/C++ from source yourself using the **Debug** configuration.

3. From the main menu select **Project**, **Properties**. This can also be accessed using the hot key **ALT** + **F7**.
4. Under **Configuration Properties**, open the tree view.
5. Select **C/C++**, **General** in the tree view.
6. In the **Additional Include Directories** text field, add the `include/` directory of Connector/C++ (it should be located within the Connector/C++ installation directory). Also add the Boost library's root directory, if Boost is required to build the application (see Section 5.1, “Building Connector/C++ Applications: General Considerations”).
7. In the tree view, open **Linker**, **General**, **Additional Library Directories**.
8. In the **Additional Library Directories** text field, add the Connector/C++ library directory (it should be located within the Connector/C++ installation directory). The library directory name ends with `lib64` (for 64-bit builds) or `lib` (for 32-bit builds).

The remaining steps depend on whether you are building an application to use the Connector/C++ dynamic or static library.

Dynamic Build

If you are building an application to use the Connector/C++ dynamic library, follow these steps:

1. Open **Linker**, **Input** in the **Property Pages** dialog.
2. Add the appropriate import library name into the **Additional Dependencies** text field (see Linking Connector/C++ to Applications).
3. Choose the C++ Runtime Library to link to. In the **Property Pages** dialog, open **C++**, **Code Generation** in the tree view, and then select the appropriate option for **Runtime Library** following these rules:

- *For version 1.1.9 and later*: Link to the dynamic version of the C++ Runtime Library by selecting the `/MD` compiler option. Also, target hosts running the client application must have the Visual C++ Redistributable for Visual Studio installed. The required version is VC++ Redistributable 2015 for Connector/C++ 1.1.11 and higher (Commercial and Community builds), VC++ Redistributable 2013 for Connector/C++ versions prior to 1.1.11 (Community builds only).

- *For version 1.1.8 and before*: Link to the dynamic version of the C++ Runtime Library by selecting the `/MD` compiler option.

Do *not* use the `/MTd` or `/MDd` option if you are using an Oracle-built Connector/C++. For an explanation, see this discussion: Connector/C++ and Application Build Configuration Must Match.

4. Copy the appropriate dynamic library to the same directory as the application executable (see Linking Connector/C++ to Applications). Alternatively, extend the `PATH` environment variable using `SET PATH=%PATH%;C:\path\to\cpp`, or copy the dynamic library to the Windows installation directory, typically `C:\windows`.

   The dynamic library must be in the same directory as the application executable, or somewhere on the system's path, so that the application can access the Connector/C++ dynamic library at runtime.

Static Build

If you are building your application to use the static library, follow these steps:

1. Open **Linker**, **Input** in the **Property Pages** dialog.

2. Add the appropriate static library name into the **Additional Dependencies** text field (see Linking Connector/C++ to Applications).

3. To compile code that is linked statically with the connector library, define a macro that adjusts API declarations in the header files for usage with the static library. By default, the macro is defined to declare functions to be compatible with an application that calls a DLL.

   In the **Project**, **Properties** tree view, under **C++**, **Preprocessor**, enter the appropriate macro into the **Preprocessor Definitions** text field: Define the `CPPCONN_PUBLIC_FUNC` macro as an empty string. To ensure this, define the macro as `CPPCONN_PUBLIC_FUNC=`, not as `CPPCONN_PUBLIC_FUNC`.

4. Choose the C++ Runtime Library to link to. In the **Property Pages** dialog, open **C++**, **Code Generation** in the tree view, and then select the appropriate option for **Runtime Library** following these rules:

   - *For version 1.1.9 and later*: Link to the dynamic version of the C++ Runtime Library by selecting the `/MD` compiler option. Also, target hosts running the client application must have the Visual C++ Redistributable for Visual Studio installed. The required version is VC++ Redistributable 2015 for Connector/C++ 1.1.11 and higher (Commercial and Community builds), VC++ Redistributable 2013 for Connector/C++ versions prior to 1.1.11 (Community builds only).

   - *For version 1.1.8 and before*: Link to the dynamic version of the C++ Runtime Library by selecting the `/MD` compiler option.

   Do *not* use the `/MTd` or `/MDd` option if you are using an Oracle-built Connector/C++. For an explanation, see this discussion: Connector/C++ and Application Build Configuration Must Match.

5.3 Building Connector/C++ 1.1 Applications on Linux with NetBeans

This section describes how to build Connector/C++ 1.1 applications for Linux using the NetBeans IDE. For general application-building information, see Section 5.1, “Building Connector/C++ Applications: General Considerations”. (These instructions have not been tested for Connector/C++ 8.0.)

1. Create a new project. Select **File**, **New Project**. Choose a **C/C++ Application** and click **Next**.
2. Give the project a name and click **Finish**. A new project is created.
3. In the **Projects** tab, right-click **Source Files** and select **New**, then **Main C++ File...**.
4. Change the filename, or simply select the defaults and click **Finish** to add the new file to the project.
5. Now add some working code to your main source file. Explore your Connector/C++ installation and navigate to the `examples` directory.
6. Select a suitable example, such as `standalone_example_docs1.cpp`. Copy all the code in this file, and use it to replace the code in your existing main source file. Amend the code to reflect the connection properties required for your test database. You now have a working example that will access a MySQL database using Connector/C++.
7. At this point, NetBeans shows some errors in the source code. Direct NetBeans to the necessary header files to include. Select **File**, **Project Properties** from the main menu.
8. In the **Categories:** tree view panel, navigate to **Build**, **C++ Compiler**.
9. In the **General** panel, select **Include Directories**.
10. Click the **...** button.
11. Click **Add**, then navigate to the directory where the Connector/C++ header files are located. This is `/usr/local/include` unless you have installed the files to a different location. Click **Select**. Click **OK**.
12. Click **OK** again to close the **Project Properties** dialog.

At this point, you have created a NetBeans project containing a single C++ source file. You have also ensured that the necessary include files are accessible. Before continuing, decide whether your project is to use the Connector/C++ dynamic or static library. The project settings are slightly different in each case, because you link against a different library.

Using the Dynamic Library

To use the Connector/C++ dynamic library, link your project with a single library file, `libmysqlcppconn.so`. The location of this file depends on how you configured your installation of Connector/C++, but typically is `/usr/local/lib`.

1. Set the project to link the necessary library file. Select **File**, **Project Properties** from the main menu.
2. In the **Categories:** tree view, navigate to **Linker**.
3. In the **General** panel, select **Additional Library Directories**. Click the **...** button.
4. Select and add the `/usr/local/lib` directories.
5. In the same panel, add the library file required for static linking as discussed earlier.
6. Click OK to close the Project Properties dialog.

After configuring your project, build it by selecting **Run**, **Build Main Project** from the main menu. You then run the project using **Run**, **Run Main Project**.

On running the application, you should see a screen that displays information like this:

```
Running 'SELECT 'Hello World!' AS _message' ...
        ... MySQL replies: Hello World!
        ... MySQL says it again: Hello World!

[Press Enter to close window]
```

> **Note**
>
> The preceding settings and procedures were carried out for the default `Debug` configuration. To create a `Release` configuration, select that configuration before setting the Project Properties.

Using the Static Library

To use the Connector/C++ static library, link against two library files, `libmysqlcppconn-static.a` and `libmysqlclient.a`. The locations of the files depend on your setup, but typically the former are in `/usr/local/lib` and the latter in `/usr/lib`. The file `libmysqlclient.a` is not part of Connector/C++, but is the MySQL client library file distributed with MySQL. (Remember, the MySQL client library is an optional component as part of the MySQL installation process.)

1. Set the project to link the necessary library files. Select **File**, **Project Properties** from the main menu.
2. In the **Categories:** tree view, navigate to **Linker**.
3. In the **General** panel, select **Additional Library Directories**. Click the **...** button.
4. Select and add the `/usr/lib` and `/usr/local/lib` directories.
5. In the same panel, add the two library files required for static linking as discussed earlier.
6. Click **OK** to close the **Project Properties** dialog.

Chapter 6 Getting Started with Connector/C++: Usage Examples

Table of Contents

Source distributions of Connector/C++ include an `examples` directory that contains usage examples that explain how to use 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 metadata

Several examples in this document are only code snippets, not complete programs. These snippets provide a brief overview on the API. For complete programs, check the `examples` directory of your Connector/C++ installation Please also read the `README` file in that directory. To test the example code, edit the `examples.h` file in the `examples` directory to add your connection information, then rebuild the code by issuing a `make` command.

The example programs in the `examples` directory include:

- `connect.cpp`:

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

- `connection_meta_schemaobj.cpp`:

  How to obtain metadata associated with a connection object, such as a list of tables or databases, the MySQL version, or the connector version.

- `debug_output.cpp`:

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

- `exceptions.cpp`:

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

- `prepared_statements.cpp`:

  How to execute Prepared Statements, including an example showing how to handle SQL statements that cannot be prepared by the MySQL server.

- `resultset.cpp`:

  How to use a cursor to fetch data and iterate over a result set.

- `resultset_meta.cpp`:

  How to obtain metadata associated with a result set, such as the number of columns and column types.

- `resultset_types.cpp`:

  Result sets returned from metadata methods. (This is more a test than an example.)

- `standalone_example.cpp`:

  Simple standalone program not integrated into regular `CMake` builds.

- `statements.cpp`:

  How to execute SQL statements without using Prepared Statements.

- `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.

6.1 Connecting to MySQL

To establish a connection to the MySQL server, retrieve an instance of `sql::Connection` from a `sql::mysql::MySQL_Driver` object. A `sql::mysql::MySQL_Driver` object is returned by `sql::mysql::get_mysql_driver_instance()`.

```
sql::mysql::MySQL_Driver *driver;
sql::Connection *con;

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

delete con;
```

Make sure that you free `con`, the `sql::Connection` object, as soon as you do not need it any more. But do not explicitly free `driver`, the connector object. Connector/C++ takes care of freeing that.

> **Note**
>
> `get_mysql_driver_instance()` calls `get_driver_instance()`, which is not thread-safe. Either avoid invoking these methods from within multiple threads at once, or surround the calls with a mutex to prevent simultaneous execution in multiple threads.

These methods can be used to check the connection state or reconnect:

- `sql::Connection::isValid()` checks whether the connection is alive
- `sql::Connection::reconnect()` reconnects if the connection has gone down

For more information about connection options, see Chapter 10, *Connector/C++ Connection Options*.

6.2 Running a Simple Query

To run simple queries, you can use the `sql::Statement::execute()`, `sql::Statement::executeQuery()`, and `sql::Statement::executeUpdate()` methods. Use the method `sql::Statement::execute()` 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 information.

```
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**
>
> You must free the `sql::Statement` and `sql::Connection` objects explicitly using `delete`.

6.3 Fetching Results

The API for fetching result sets is identical for (simple) statements and prepared statements. If your query returns one result set, use `sql::Statement::executeQuery()` or `sql::PreparedStatement::executeQuery()` to run your query. Both methods return `sql::ResultSet` objects. By default, Connector/C++ buffers 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 = " << res->getInt(1); // getInt(1) returns the first column
  // ... or column names for accessing results.
  // The latter is recommended.
  cout << ", label = '" << res->getString("label") << "'" << endl;
}

delete res;
```

```
delete stmt;
delete con;
```

> **Note**
>
> In the preceding code snippet, column indexing starts from 1.

> **Note**
>
> You must free the `sql::Statement`, `sql::Connection`, and `sql::ResultSet` objects explicitly using `delete`.

Cursor usage is demonstrated in the examples contained in the download package.

6.4 Using Prepared Statements

If you are not familiar with Prepared Statements in MySQL, take a look at the source code comments and explanations in the file `examples/prepared_statement.cpp`.

`sql::PreparedStatement` is created by passing an 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;
```

> **Note**
>
> You must free the `sql::PreparedStatement` and `sql::Connection` objects explicitly using `delete`.

6.5 Complete Example 1

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

```
/* Copyright 2008, 2010, Oracle and/or its affiliates. All rights reserved.

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., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 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 data 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;
}
```

6.6 Complete Example 2

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

```
/* Copyright 2008, 2010, Oracle and/or its affiliates. All rights reserved.

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., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 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 7 Connector/C++ Tutorials

Table of Contents

The following tutorials illustrate various aspects of using MySQL Connector/C++. Also consult the examples in Chapter 6, *Getting Started with Connector/C++: Usage Examples*.

7.1 Prerequisites and Background Information

This section describes the prerequisites that must be satisifed before you work through the remaining tutorial sections, and shows how to set up the framework code that serves as the basis for the tutorial applications.

These tutorials refer to tables and sample data from the `world` database, which you can download from the “Example Databases” section of the MySQL Documentation page.

Each tutorial application uses a framework consisting of the following code. The examples vary at the line that says `/* INSERT TUTORIAL CODE HERE! */` within the `try` block, which is replaced for each application with the application-specific code.

```
#include <stdlib.h>
#include <iostream>
#include <sstream>
#include <stdexcept>
/* uncomment for applications that use vectors */
/*#include <vector>*/

#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>

#define EXAMPLE_HOST "localhost"
#define EXAMPLE_USER "worlduser"
#define EXAMPLE_PASS "worldpass"
#define EXAMPLE_DB "world"

using namespace std;

int main(int argc, const char **argv)
{
  string url(argc >= 2 ? argv[1] : EXAMPLE_HOST);
```

```
  const string user(argc >= 3 ? argv[2] : EXAMPLE_USER);
  const string pass(argc >= 4 ? argv[3] : EXAMPLE_PASS);
  const string database(argc >= 5 ? argv[4] : EXAMPLE_DB);

  cout << "Connector/C++ tutorial framework..." << endl;
  cout << endl;

  try {

    /* INSERT TUTORIAL CODE HERE! */

  } catch (sql::SQLException &e) {
    /*
      MySQL Connector/C++ throws three different exceptions:

      - sql::MethodNotImplementedException (derived from sql::SQLException)
      - sql::InvalidArgumentException (derived from sql::SQLException)
      - sql::SQLException (derived from std::runtime_error)
    */
    cout << "# ERR: SQLException in " << __FILE__;
    cout << "(" << __FUNCTION__ << ") on line " << __LINE__ << endl;
    /* what() (derived from std::runtime_error) fetches error message */
    cout << "# ERR: " << e.what();
    cout << " (MySQL error code: " << e.getErrorCode();
    cout << ", SQLState: " << e.getSQLState() << " )" << endl;

    return EXIT_FAILURE;
  }

  cout << "Done." << endl;
  return EXIT_SUCCESS;
}
```

To try the framework code as a standalone program, use this procedure:

1. Copy and paste the framework code to a file such as `framework.cpp`. Edit the `#define` statements to reflect your connection parameters (server, user, password, database). Also, because the file contains those parameters, set its access mode to be readable only to yourself.

2. Compile the framework. For example, on macOS, the command might look like this (enter the command on one line):

   ```
   shell> g++ -o framework
     -I/usr/local/include -I/usr/local/include/cppconn
     framework.cpp -lmysqlcppconn
   ```

   Adapt the command as necessary for your system. A similar command is needed for the tutorial applications that follow.

3. To run the framework, enter the following:

   ```
   shell> ./framework
   ```

   You will see a simple message:

   ```
   Connector/C++ tutorial framework...

   Done.
   ```

You are now ready to continue to the tutorials.

7.2 Calling Stored Procedures with `Statement` Objects

A stored procedure can be called using a `Statement` or `PreparedStatement` object. This section shows how to call stored procedures using `Statement` objects. To see how to use `PreparedStatement` objects, see Section 7.3, “Calling Stored Procedures with `PreparedStatement` Objects”.

The following list describes different types of stored procedures that you can construct and call, along with example stored procedures that illustrate each type:

1. A stored procedure that returns no result. For example, such a stored procedure can log non-critical information, or change database data in a straightforward way.

   The following procedure adds a country to the `world` database, but does not return a result:

   ```
   CREATE PROCEDURE add_country (IN country_code CHAR(3),
                                 IN country_name CHAR(52),
                                 IN continent_name CHAR(30))
   BEGIN
     INSERT INTO Country(Code, Name, Continent)
       VALUES (country_code, country_name, continent_name);
   END;
   ```

2. A stored procedure that returns one or more values using output parameters. For example, such a procedure can indicate success or failure, or retrieve and return data items.

   The following procedures use an output parameter to return the population of a specified country or continent, or the entire world:

   ```
   CREATE PROCEDURE get_pop (IN country_name CHAR(52),
                             OUT country_pop BIGINT)
   BEGIN
     SELECT Population INTO country_pop FROM Country
       WHERE Name = country_name;
   END;
   ```

   ```
   CREATE PROCEDURE get_pop_continent (IN continent_name CHAR(30),
                                       OUT continent_pop BIGINT)
   BEGIN
     SELECT SUM(Population) INTO continent_pop FROM Country
       WHERE Continent = continent_name;
   END;
   ```

   ```
   CREATE PROCEDURE get_pop_world (OUT world_pop BIGINT)
   BEGIN
     SELECT SUM(Population) INTO world_pop FROM Country;
   END;
   ```

3. A stored procedure that returns one or more result sets. The procedure can execute one or more queries, each of which returns an arbitrary number of rows. Your application loops through each result set to display, transform, or otherwise process each row in it.

   This procedure returns several result sets:

   ```
   CREATE PROCEDURE get_data ()
   BEGIN
     SELECT Code, Name, Population, Continent FROM Country
       WHERE Continent = 'Oceania' AND Population < 10000;
   ```

```
  SELECT Code, Name, Population, Continent FROM Country
    WHERE Continent = 'Europe' AND Population < 10000;
  SELECT Code, Name, Population, Continent FROM Country
    WHERE Continent = 'North America' AND Population < 10000;
END;
```

Enter and test the stored procedures manually to ensure that they will be available to your C++ applications. (Select world as the default database before you create them.) You are now ready to start writing applications using Connector/C++ that call stored procedures.

7.2.1 Using a Statement for a Stored Procedure That Returns No Result

This example shows how to call a stored procedure that returns no result set.

1. Make a copy of the tutorial framework code:

```
shell> cp framework.cpp sp_scenario1.cpp
```

2. Add the following code to the try block of the tutorial framework:

```
sql::Driver* driver = get_driver_instance();
std::auto_ptr<sql::Connection> con(driver->connect(url, user, pass));
con->setSchema(database);
std::auto_ptr<sql::Statement> stmt(con->createStatement());

// We need not check the return value explicitly. If it indicates
// an error, Connector/C++ generates an exception.
stmt->execute("CALL add_country('ATL', 'Atlantis', 'North America')");
```

3. Compile the program as described in Section 7.1, “Prerequisites and Background Information”.

4. Run the program:

```
shell> ./sp_scenario1
```

5. Using the mysql command-line client or other suitable program, check the world database to determine that it has been updated correctly. You can use this query:

```
mysql> SELECT Code, Name, Continent FROM Country WHERE Code='ATL';
+------+----------+---------------+
| Code | Name     | Continent     |
+------+----------+---------------+
| ATL  | Atlantis | North America |
+------+----------+---------------+
```

The code in this application simply invokes the execute method, passing to it a statement that calls the stored procedure. The procedure itself returns no value, although it is important to note there is always a return value from the CALL statement; this is the execute status. Connector/C++ handles this status for you, so you need not handle it explicitly. If the execute call fails for some reason, it raises an exception that the catch block handles.

7.2.2 Using a Statement for a Stored Procedure That Returns an Output Parameter

This example shows how to handle a stored procedure that returns an output parameter.

1. Make a copy of the tutorial framework code:

```
shell> cp framework.cpp sp_scenario2.cpp
```

2. Add the following code to the `try` block of the tutorial framework:

```
sql::Driver* driver = get_driver_instance();
std::auto_ptr<sql::Connection> con(driver->connect(url, user, pass));
con->setSchema(database);
std::auto_ptr<sql::Statement> stmt(con->createStatement());

stmt->execute("CALL get_pop('Uganda', @pop)");

std::auto_ptr<sql::ResultSet> res(stmt->executeQuery("SELECT @pop AS _reply"));
while (res->next())
  cout << "Population of Uganda: " << res->getString("_reply") << endl;

stmt->execute("CALL get_pop_continent('Asia', @pop)");

res.reset(stmt->executeQuery("SELECT @pop AS _reply"));
while (res->next())
  cout << "Population of Asia: " << res->getString("_reply") << endl;

stmt->execute("CALL get_pop_world(@pop)");

res.reset(stmt->executeQuery("SELECT @pop AS _reply"));
while (res->next())
  cout << "Population of World: " << res->getString("_reply") << endl;
```

3. Compile the program as described in Section 7.1, “Prerequisites and Background Information”.

4. Run the program:

```
shell> ./sp_scenario2
Connector/C++ tutorial framework...

Population of Uganda: 21778000
Population of Asia: 3705025700
Population of World: 6078749450
Done.
```

In this scenario, each stored procedure sets the value of an output parameter. This is not returned directly to the `execute` method, but needs to be obtained using a subsequent query. If you were executing the SQL statements directly, you might use statements similar to these:

```
CALL get_pop('Uganda', @pop);
SELECT @pop;
CALL get_pop_continent('Asia', @pop);
SELECT @pop;
CALL get_pop_world(@pop);
SELECT @pop;
```

In the C++ code, a similar sequence is carried out for each procedure call:

1. Execute the `CALL` statement.

2. Obtain the output parameter by executing an additional query. The query produces a `ResultSet` object.

3. Retrieve the data using a `while` loop. The simplest way to do this is to use a `getString` method on the `ResultSet`, passing the name of the variable to access. In this example `_reply` is used as a placeholder for the variable and therefore is used as the key to access the correct element of the result dictionary.

   Although the query used to obtain the output parameter returns only a single row, it is important to use the `while` loop to catch more than one row, to avoid the possibility of the connection becoming unstable.

7.2.3 Using a `Statement` for a Stored Procedure That Returns a Result Set

This example shows how to handle result sets produced by a stored procedure.

1. Make a copy of the tutorial framework code:

```
shell> cp framework.cpp sp_scenario3.cpp
```

2. Add the following code to the `try` block of the tutorial framework:

```
sql::Driver* driver = get_driver_instance();
std::auto_ptr<sql::Connection> con(driver->connect(url, user, pass));
con->setSchema(database);
std::auto_ptr<sql::Statement> stmt(con->createStatement());

stmt->execute("CALL get_data()");
std::auto_ptr< sql::ResultSet > res;
do {
  res.reset(stmt->getResultSet());
  while (res->next()) {
    cout << "Name: " << res->getString("Name")
         << " Population: " << res->getInt("Population")
         << endl;
  }
} while (stmt->getMoreResults());
```

3. Compile the program as described in Section 7.1, “Prerequisites and Background Information”.

4. Run the program:

```
shell> ./sp_scenario3
Connector/C++ tutorial framework...

Name: Cocos (Keeling) Islands Population: 600
Name: Christmas Island Population: 2500
Name: Norfolk Island Population: 2000
Name: Niue Population: 2000
Name: Pitcairn Population: 50
Name: Tokelau Population: 2000
Name: United States Minor Outlying Islands Population: 0
Name: Svalbard and Jan Mayen Population: 3200
Name: Holy See (Vatican City State) Population: 1000
Name: Anguilla Population: 8000
Name: Atlantis Population: 0
Name: Saint Pierre and Miquelon Population: 7000
Done.
```

The code is similar to the examples shown previously. The code of particular interest here is:

```
do {
  res.reset(stmt->getResultSet());
  while (res->next()) {
    cout << "Name: " << res->getString("Name")
         << " Population: " << res->getInt("Population")
         << endl;
  }
} while (stmt->getMoreResults());
```

The CALL is executed as before, but this time the results are returned into multiple ResultSet objects because the stored procedure executes multiple SELECT statements. In this example, the output shows that three result sets are processed, because there are three SELECT statements in the stored procedure. Each result set returns more than one row.

The results are processed using this code pattern:

```
do {
  Get Result Set
  while (Get Result) {
    Process Result
  }
} while (Get More Result Sets);
```

> **Note**
>
> Use this pattern even if the stored procedure executes only a single SELECT and produces only one result set. This is a requirement of the underlying protocol.

7.3 Calling Stored Procedures with PreparedStatement Objects

This section shows how to call stored procedures using prepared statements. It is recommended that, before working through it, you first work through the previous tutorial Section 7.2, “Calling Stored Procedures with Statement Objects”. That section shows the stored procedures required by the applications in this section.

7.3.1 Using a PreparedStatement for a Stored Procedure That Returns No Result

This example shows how to call a stored procedure that returns no result set.

1. Make a copy of the tutorial framework code:

   ```
   shell> cp framework.cpp ps_scenario1.cpp
   ```

2. Add the following code to the try block of the tutorial framework:

   ```
   vector<string> code_vector;
   code_vector.push_back("SLD");
   code_vector.push_back("DSN");
   code_vector.push_back("ATL");

   vector<string> name_vector;
   ```

```
name_vector.push_back("Sealand");
name_vector.push_back("Disneyland");
name_vector.push_back("Atlantis");

vector<string> cont_vector;
cont_vector.push_back("Europe");
cont_vector.push_back("North America");
cont_vector.push_back("Oceania");

sql::Driver * driver = get_driver_instance();

std::auto_ptr< sql::Connection > con(driver->connect(url, user, pass));
con->setSchema(database);

std::auto_ptr< sql::PreparedStatement >  pstmt;

pstmt.reset(con->prepareStatement("CALL add_country(?,?,?)"));
for (int i=0; i<3; i++)
{
  pstmt->setString(1,code_vector[i]);
  pstmt->setString(2,name_vector[i]);
  pstmt->setString(3,cont_vector[i]);

  pstmt->execute();
}
```

Also, uncomment `#include <vector>` near the top of the code, because vectors are used to store sample data.

3. Compile the program as described in Section 7.1, “Prerequisites and Background Information”.

4. Run the program:

```
shell> ./ps_scenario1
```

5. You can check whether the database has been updated correctly by using this query:

```
mysql> SELECT Code, Name, Continent FROM Country
    -> WHERE Code IN('DSN','ATL','SLD');
+------+------------+---------------+
| Code | Name       | Continent     |
+------+------------+---------------+
| ATL  | Atlantis   | Oceania       |
| DSN  | Disneyland | North America |
| SLD  | Sealand    | Europe        |
+------+------------+---------------+
```

The code is relatively simple, as no processing is required to handle result sets. The procedure call, `CALL add_country(?,?,?)`, is made using placeholders for input parameters denoted by `'?'`. These placeholders are replaced by the appropriate data values using the `PreparedStatement` object's `setString` method. The `for` loop is set up to iterate 3 times, as there are three data sets in this example. The same `PreparedStatement` is executed three times, each time with different input parameters.

7.3.2 Using a `PreparedStatement` for a Stored Procedure That Returns an Output Parameter

This example shows how to handle a stored procedure that returns an output parameter.

1. Make a copy of the tutorial framework code:

```
shell> cp framework.cpp ps_scenario2.cpp
```

2. Add the following code to the try block of the tutorial framework:

```
vector<string> cont_vector;
cont_vector.push_back("Europe");
cont_vector.push_back("North America");
cont_vector.push_back("Oceania");

sql::Driver * driver = get_driver_instance();

std::auto_ptr< sql::Connection > con(driver->connect(url, user, pass));
con->setSchema(database);

std::auto_ptr< sql::Statement > stmt(con->createStatement());
std::auto_ptr< sql::PreparedStatement >  pstmt;
std::auto_ptr< sql::ResultSet > res;

pstmt.reset(con->prepareStatement("CALL get_pop_continent(?,@pop)"));

for (int i=0; i<3; i++)
{
  pstmt->setString(1,cont_vector[i]);
  pstmt->execute();
  res.reset(pstmt->executeQuery("SELECT @pop AS _population"));
  while (res->next())
    cout << "Population of "
         << cont_vector[i]
         << " is "
         << res->getString("_population") << endl;
}
```

   Also, uncomment `#include <vector>` near the top of the code, because vectors are used to store sample data.

3. Compile the program as described in Section 7.1, “Prerequisites and Background Information”.

4. Run the program:

```
shell> ./ps_scenario2
Connector/C++ tutorial framework...

Population of Europe is 730074600
Population of North America is 482993000
Population of Oceania is 30401150
Done.
```

In this scenario a PreparedStatement object is created that calls the get_pop_continent stored procedure. This procedure takes an input parameter, and also returns an output parameter. The approach used is to create another statement that can be used to fetch the output parameter using a SELECT query. Note that when the PreparedStatement is created, the input parameter to the stored procedure is denoted by '?'. Prior to execution of the prepared statement, it is necessary to replace this placeholder by an actual value. This is done using the setString method:

```
pstmt->setString(1,cont_vector[i]);
```

Although the query used to obtain the output parameter returns only a single row, it is important to use the while loop to catch more than one row, to avoid the possibility of the connection becoming unstable.

7.3.3 Using a PreparedStatement for a Stored Procedure That Returns a Result Set

This example shows how to handle result sets produced by a stored procedure.

1. Make a copy of the tutorial framework code:

```
shell> cp framework.cpp ps_scenario3.cpp
```

2. Add the following code to the try block of the tutorial framework:

```
sql::Driver * driver = get_driver_instance();

std::auto_ptr< sql::Connection > con(driver->connect(url, user, pass));
con->setSchema(database);

std::auto_ptr< sql::PreparedStatement >  pstmt;
std::auto_ptr< sql::ResultSet > res;

pstmt.reset(con->prepareStatement("CALL get_data()"));
res.reset(pstmt->executeQuery());

for(;;)
{
  while (res->next()) {
    cout << "Name: " << res->getString("Name")
         << " Population: " << res->getInt("Population")
         << endl;
  }
  if (pstmt->getMoreResults())
  {
    res.reset(pstmt->getResultSet());
    continue;
  }
  break;
}
```

3. Compile the program as described in Section 7.1, “Prerequisites and Background Information”.

4. Run the program:

```
shell> ./ps_scenario3
```

5. Make a note of the output generated.

The code executes the stored procedure using a PreparedStatement object. The standard do/while construct is used to ensure that all result sets are fetched. The returned values are fetched from the result sets using the getInt and getString methods.

Chapter 8 Connector/C++ Debug Tracing

Although a debugger can be used to debug a specific instance of your application, you may find it beneficial to enable 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.

Connector/C++ can write two trace files:

1. A trace file generated by the MySQL client library
2. A trace file generated internally by Connector/C++

The first trace file can be generated by the underlying MySQL client library (`libmysqlclient`). To enable this trace, the connector calls the `mysql_debug()` C API function internally. Because only debug versions of the MySQL client library are capable of writing a trace file, compile Connector/C++ against a debug version of the library to use this trace. The trace shows the internal function calls and the addresses of internal objects as shown here:

```
>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 Connector/C++ internal trace. It is available with debug and nondebug builds of the connector as long as you have enabled the tracing module at compile time. By default, tracing functionality is not available and calls to trace functions are removed by the preprocessor. To enable the tracing module, configure Connector/C++ with the `-DMYSQLCPPCONN_TRACE_ENABLE=1` `CMake` option.

Compiling the connector with tracing functionality enabled causes two additional tracing function calls per each connector function call. For example:

```
|  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
[...]
```

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 runtime 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 I/O 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 runtime of your application is not impacted significantly.

The example from `examples/debug_output.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 `setClientOption()` method of a connection object to activate and deactivate trace generation. The MySQL client library trace always writes its trace to a file, whereas the connector writes protocol messages to the standard output.

```
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");

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

Chapter 9 Connector/C++ Usage Notes

Connector/C++ is compatible with the JDBC 4.0 API. See the JDBC overview for information on JDBC 4.0. Please also check the `examples` directory of the download package.

- The Connector/C++ `sql::DataType` class 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`, `SQLNULL`.

  Connector/C++ does not support the following JDBC standard data types: `ARRAY`, `BLOB`, `CLOB`, `DISTINCT`, `FLOAT`, `OTHER`, `REF`, `STRUCT`.

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

- Two non-JDBC methods let you fetch and set unsigned integers: `getUInt64()` and `getUInt()`. These are available for `ResultSet` and `Prepared_Statement`:

  - `ResultSet::getUInt64()`
  - `ResultSet::getUInt()`
  - `Prepared_Statement::setUInt64()`
  - `Prepared_Statement::setUInt()`

- The `DatabaseMetaData::getColumns()` method 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`: A string which is “YES” if the column is an auto-increment column, “NO” otherwise.

- Connector/C++ may return different metadata for the same column, depending on the method you call.

  Suppose that you have a column that accepts a character set and a collation in its specification and you specify a binary collation, such as:

```
VARCHAR(20) CHARACTER SET utf8 COLLATE utf8_bin
```

  The server sets the `BINARY` flag in the result set metadata of this column. The `ResultSetMetaData::getColumnTypeName()` method uses the metadata and reports, due to the `BINARY` flag, that the column type name is `BINARY`, as illustrated here:

```
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` and will report the type name `VARCHAR` for the same column. It also returns a different type code.

- When inserting or updating `BLOB` or `TEXT` columns, Connector/C++ developers are advised not to use `setString()`. Instead, use the dedicated `setBlob()` API function.

  The use of `setString()` can cause a Packet too large error message. The error occurs 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 Connector/C++, because it could lead to security issues, such as extremely large memory allocation requests due to malevolently long strings.

  If `setBlob()` is used, this problem does not arise because `setBlob()` takes a streaming approach based on `std::istream`. When sending the data from the stream to the MySQL server, Connector/C++ splits the stream into chunks appropriate for the server using the current `max_allowed_packet` setting.

  > **Caution**
  >
  > When using `setString()`, it is not possible to set `max_allowed_packet` to a value large enough for the string prior to passing it to Connector/C++. That configuration option cannot be changed within a session.

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

- In general, 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 is 5.6. Connector/C++ is primarily targeted at the MySQL GA version that is available on its release.

The following methods throw a `sql::MethodNotImplemented` exception when you connect to a MySQL server earlier than 5.1:

- `DatabaseMetaData::getCrossReference()`
- `DatabaseMetaData::getExportedKeys()`

- Connector/C++ includes a `Connection::getClientOption()` method that 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 `metadataUseInfoSchema` connection option controls whether to use the `Information_Schemata` for returning the metadata of `SHOW` statements:

- For `metadataUseInfoSchema`, interpret the `optionValue` argument as a boolean upon return.
- For `defaultStatementResultType` and `defaultPreparedStatementResultType`, interpret the `optionValue` argument as an integer upon return.

The connection property can be set either when establishing the connection through the connection property map, or using `void Connection::setClientOption(const std::string & optionName, const void * optionValue)` where `optionName` is assigned the value `metadataUseInfoSchema`.

Some examples:

```
bool isInfoSchemaUsed;
conn->getClientOption("metadataUseInfoSchema", (void *) &isInfoSchemaUsed);

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

- To get and set MySQL session variables, Connector/C++ supports the following `MySQL_Connection` methods, which are 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)
```

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

```
SHOW SESSION VARIABLES LIKE 'var_name'
```

You can use the % and _ SQL pattern characters in *var_name*.

`setSessionVariable()` is equivalent to executing:

```
SET SESSION var_name = value
```

- Fetching the value of a column can sometimes return different values depending on whether the call is made from a Statement or Prepared Statement. This is because the protocol used to communicate with the server differs depending on whether a Statement or Prepared Statement is used.

  To illustrate this, consider the case where a column has been defined as type `BIGINT`. The most negative `BIGINT` value is then inserted into the column. If a Statement and Prepared Statement are created that perform a `getUInt64()` call, then the results will be different in each case. The Statement returns the maximum positive value for `BIGINT`. The Prepared Statement returns 0.

  The difference results from the fact that Statements use a text protocol, and Prepared Statements use a binary protocol. With the binary protocol in this case, a binary value is returned from the server that can be interpreted as an `int64`. In the preceding scenario, a very large negative value is fetched with `getUInt64()`, which fetches unsigned integers. Because the large negative value cannot be sensibly converted to an unsigned value, 0 is returned.

  In the case of the Statement, which uses the text protocol, values are returned from the server as strings, and then converted as required. When a string value is returned from the server in the preceding scenario, the large negative value must be converted by the runtime library function `strtoul()`, which `getUInt64()` calls. The behavior of `strtoul()` is dependent upon the specific runtime and host operating system, so the results can be platform dependent. In the case, given a large positive value was actually returned.

  Although it is very rare, there are some cases where Statements and Prepared Statements can return different values unexpectedly, but this usually only happens in extreme cases such as the one mentioned.

- The JDBC documentation lists many fields for the `DatabaseMetaData` class. JDBC also appears to define certain values for those fields. However, Connector/C++ does not define certain values for those fields. Internally enumerations are used and the compiler determines the values to assign to a field.

  To compare a value with the field, use code such as the following, rather than making assumptions about specific values for the attribute:

  ```
  // dbmeta is an instance of DatabaseMetaData
  if (myvalue == dbmeta->attributeNoNulls) {
      ...
  }
  ```

  Usually `myvalue` will be a column from a result set holding metadata information. Connector/C++ does not guarantee that `attributeNoNulls` is 0. It can be any value.

- When programming stored procedures, JDBC has available an extra class, an extra abstraction layer for callable statements, the `CallableStatement` class. As this class is not present in Connector/C++, use the methods from the `Statement` and `PreparedStatement` classes to execute a stored procedure using `CALL`.

Chapter 10 Connector/C++ Connection Options

To connect to a MySQL server from Connector/C++ applications, use the `connect()` method of the `MySQL_Driver` class. The `connect()` method has two calling sequences:

- One calling sequence takes arguments indicating how to connect to the MySQL server, and the user name and password of the MySQL account to use:

```
sql::Connection * MySQL_Driver::connect(const sql::SQLString& hostName,
                                        const sql::SQLString& userName,
                                        const sql::SQLString& password)
```

  Example:

```
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", "root", "rootpass");
```

  This syntax is convenient for simple connections.

- The other syntax takes an option map that contains the connection properties to use for establishing the connection:

```
sql::Connection * MySQL_Driver::connect(sql::ConnectOptionsMap & properties)
```

  This syntax is useful for connections that require specifying options other than the three permitted by the first syntax. To use an option map, initialize it with the required connection properties, then pass the map to the `connect()` call.

  Example:

```
sql::mysql::MySQL_Driver *driver;
sql::Connection *con;
sql::ConnectOptionsMap connection_properties;

connection_properties["hostName"] = hostName;
connection_properties["userName"] = userName;
connection_properties["password"] = password;
connection_properties["schema"] = "information_schema";
connection_properties["port"] = 13306;
connection_properties["OPT_RECONNECT"] = true;

driver = sql::mysql::MySQL_Driver::get_mysql_driver_instance();
con = driver->connect(connection_properties);
```

The `hostName` parameter can be a host name, IP address, or URL.

For a host name or IP address by itself, Connector/C++ makes a TCP/IP connection to the named host. If the host name is `localhost`, Connector/C++ interprets it as `127.0.0.1`.

For a `hostName` value specified as a URL, the format begins with a connection protocol and the protocol determines the syntax of the remaining part of the URL:

- `tcp://...`

This URL format establishes a TCP/IP connection and is usable on all platforms. The format permits specification of host hame or IP address, TCP/IP port number, and default database. The syntax for a URL that includes all those items looks like this:

```
tcp://host:port/db
```

The `:port` and `/db` parts of the URL are optional. The *`host`* part may be enclosed within `[` and `]` characters, which is useful for specifying IPv6 addresses such as `::1` that contain the `:` character that otherwise would be interpreted as beginning a `:port` specifier.

This URL connects to the local host using the default port number and without selecting a default database:

```
tcp://127.0.0.1
```

This URL connects to the named host on port 13306 and selects `employees` as the default database:

```
tcp://host1.example.com:13306/employees
```

This URL connects to port 3307 on the local host, using the `::1` IPv6 address. The URL uses `[` and `]` around the address to disambiguate the `:` characters in the host and port parts of the URL:

```
tcp://[::1]:3307
```

- `pipe://pipe_name`

  This URL format enables use of named pipes for connections to the local host on Windows systems. The *`pipe_name`* value is the named pipe name, just as for the `--socket` option of MySQL clients such as `mysql` and `mysqladmin` running on Windows (see Connecting to the MySQL Server).

- `unix://path_name`

  This URL format enables use of Unix domain socket files for connections to the local host on Unix and Unix-like systems. The *`path_name`* value is the socket file path name, just as for the `--socket` option of MySQL clients such as `mysql` and `mysqladmin` running on Unix (see Connecting to the MySQL Server).

For the `connect()` syntax that takes an option map argument, Connector/C++ supports the connection properties described in the following list.

> **Note**
>
> Many of these properties correspond to arguments for the `mysql_options()`, `mysql_options4()`, or `mysql_real_connect()`, C API function. For such properties, the descriptions here are brief. For more information, refer to the descriptions for those functions. See mysql_options(), mysql_options4(), and mysql_real_connect().

- `characterSetResults`

  This option sets the `character_set_results` system variable for the session. The value is a string.

- `charsetDir`

The path name to the directory that contains character set definition files. This option corresponds to the `MYSQL_SET_CHARSET_DIR` option for the `mysql_options()` C API function. The value is a string.

This option was added in Connector/C++ 1.1.5.

- `CLIENT_COMPRESS`

  Whether to use compression in the client/server protocol. This option corresponds to the flag of the same name for the `client_flag` argument of the `mysql_real_connect()` C API function. The value is a boolean.

- `CLIENT_FOUND_ROWS`

  Whether to return the number of found (matched) rows, not the number of changed rows. This option corresponds to the flag of the same name for the `client_flag` argument of the `mysql_real_connect()` C API function. The value is a boolean.

- `CLIENT_IGNORE_SIGPIPE`

  Whether to prevent the `libmysqlclient` client library from installing a `SIGPIPE` signal handler. This option corresponds to the flag of the same name for the `client_flag` argument of the `mysql_real_connect()` C API function. The value is a boolean.

- `CLIENT_IGNORE_SPACE`

  Whether to permit spaces after function names. This option corresponds to the flag of the same name for the `client_flag` argument of the `mysql_real_connect()` C API function. The value is a boolean.

- `CLIENT_INTERACTIVE`

  Whether to permit `interactive_timeout` seconds of inactivity (rather than `wait_timeout` seconds) before closing the connection. This option corresponds to the flag of the same name for the `client_flag` argument of the `mysql_real_connect()` C API function. The value is a boolean.

- `CLIENT_LOCAL_FILES`

  Whether to enable `LOAD DATA LOCAL INFILE` handling. This option corresponds to the flag of the same name for the `client_flag` argument of the `mysql_real_connect()` C API function. The value is a boolean.

- `CLIENT_MULTI_STATEMENTS`

  Whether the client may send multiple statements in a single string (separated by `;` characters). This option corresponds to the flag of the same name for the `client_flag` argument of the `mysql_real_connect()` C API function. The value is a boolean.

  > **Note**
  >
  > There is no `CLIENT_MULTI_RESULTS` connection option. Connector/C++ enables that flag for all connections.

- `CLIENT_NO_SCHEMA`

  Whether to prohibit *`db_name.tbl_name.col_name`* syntax. This option corresponds to the flag of the same name for the `client_flag` argument of the `mysql_real_connect()` C API function. The value is a boolean.

- `defaultAuth`

The name of the authentication plugin to use. This option corresponds to the `MYSQL_DEFAULT_AUTH` option for the `mysql_options()` C API function. The value is a string.

This option was added in Connector/C++ 1.1.5.

- `defaultPreparedStatementResultType`

  The result set type for statements executed using `MySQL_Connection::prepareStatement()` to define whether result sets are scrollable. Permitted values are `sql::ResultSet::TYPE_FORWARD_ONLY` and `sql::ResultSet::TYPE_SCROLL_INSENSITIVE`. The `sql::ResultSet::TYPE_SCROLL_SENSITIVE` type is not supported.

- `defaultStatementResultType`

  The result set type for statements executed using `MySQL_Connection::createStatement()` to define whether result sets are scrollable. Permitted values are `sql::ResultSet::TYPE_FORWARD_ONLY` and `sql::ResultSet::TYPE_SCROLL_INSENSITIVE`. The `sql::ResultSet::TYPE_SCROLL_SENSITIVE` type is not supported.

- `hostName`

  This has the same meaning and syntax as for the three-argument `connect()` syntax. The value can be a host name, IP address, or URL, as described earlier in this section.

- `OPT_CAN_HANDLE_EXPIRED_PASSWORDS`

  Whether the client can handle expired passwords. This option corresponds to the `MYSQL_OPT_CAN_HANDLE_EXPIRED_PASSWORDS` option for the `mysql_options()` C API function. The value is a boolean.

  This option was added in Connector/C++ 1.1.2.

- `OPT_CHARSET_NAME`

  The name of the character set to use as the default character set. This option corresponds to the `MYSQL_SET_CHARSET_NAME` option for the `mysql_options()` C API function. The value is a string.

- `OPT_CONNECT_ATTR_ADD`

  Key-value pairs to add to the current set of connection attributes to pass to the server at connect time. This option corresponds to the `MYSQL_OPT_CONNECT_ATTR_ADD` option for the `mysql_options4()` C API function. The value is a `std::map< sql::SQLString, sql::SQLString >` value.

  This option was added in Connector/C++ 1.1.4.

- `OPT_CONNECT_ATTR_DELETE`

  Key names for key-value pairs to delete from the current set of connection attributes to pass to the server at connect time. This option corresponds to the `MYSQL_OPT_CONNECT_ATTR_DELETE` option for the `mysql_options()` C API function. The value is a `std::list< sql::SQLString >` value.

  This option was added in Connector/C++ 1.1.5.

- `OPT_CONNECT_ATTR_RESET`

Resets (clears) the current set of connection attributes to pass to the server at connect time. This option corresponds to the `MYSQL_OPT_CONNECT_ATTR_RESET` option for the `mysql_options()` C API function.

This option was added in Connector/C++ 1.1.5.

- `OPT_CONNECT_TIMEOUT`

  The connect timeout in seconds. This option corresponds to the `MYSQL_OPT_CONNECT_TIMEOUT` option for the `mysql_options()` C API function. The value is an unsigned integer.

- `OPT_ENABLE_CLEARTEXT_PLUGIN`

  Enable the `mysql_clear_password` cleartext authentication plugin. This option corresponds to the `MYSQL_ENABLE_CLEARTEXT_PLUGIN` option for the `mysql_options()` C API function. The value is a boolean.

  This option was added in Connector/C++ 1.1.3.

- `OPT_GET_SERVER_PUBLIC_KEY`

  For connections to the server made using the legacy protocol (that is, not made using X DevAPI or X DevAPI for C), Connector/C++, request the RSA public key from the server. For accounts that use the `caching_sha2_password` or `sha256_password` authentication plugin, this key can be used during the connection process for RSA key-pair based password exchange with TLS disabled. This option corresponds to the `MYSQL_OPT_GET_SERVER_PUBLIC_KEY` option for the `mysql_options()` C API function. The value is a boolean.

  This capability requires a MySQL 8.0 GA server, and is supported only for Connector/C++ built using OpenSSL.

  This option was added in Connector/C++ 1.1.11.

- `OPT_LOCAL_INFILE`

  Whether to enable the `LOAD DATA LOCAL INFILE` statement. This option corresponds to the `MYSQL_OPT_LOCAL_INFILE` option for the `mysql_options()` C API function. The value is an unsigned integer.

  This option was added in Connector/C++ 1.1.5.

- `OPT_NAMED_PIPE`

  Use a named pipe to connect to the MySQL server on Windows, if the server permits named-pipe connections. This option corresponds to the `MYSQL_OPT_NAMED_PIPE` option for the `mysql_options()` C API function. The value is unused.

- `OPT_READ_TIMEOUT`

  The timeout in seconds for each attempt to read from the server. This option corresponds to the `MYSQL_OPT_READ_TIMEOUT` option for the `mysql_options()` C API function. The value is an unsigned integer.

- `OPT_RECONNECT`

Enable or disable automatic reconnection to the server if the connection is found to have been lost. This option corresponds to the `MYSQL_OPT_RECONNECT` option for the `mysql_options()` C API function. The value is a boolean. The default is false.

- `OPT_REPORT_DATA_TRUNCATION`

  Enable or disable reporting of data truncation errors for prepared statements using the `error` member of `MYSQL_BIND` structures. This option corresponds to the `MYSQL_REPORT_DATA_TRUNCATION` option for the `mysql_options()` C API function. The value is a boolean.

- `OPT_TLS_VERSION`

  Specify the protocols permitted for encrypted connections. The option value is string containing a comma-separated list of one or more protocol names. Example:

```
connection_properties["OPT_TLS_VERSION"] = sql::SQLString("TLSv1.1,TLSv1.2");
```

  The permitted values depend on the SSL library used to compile MySQL: `TLSv1`, `TLSv1.1`, `TLSv1.2` if OpenSSL was used; `TLSv1` and `TLSv1.1` if yaSSL was used. The default is to permit all available protocols.

  For more information about connection protocols in MySQL, see Encrypted Connection Protocols and Ciphers.

  This option was added in Connector/C++ 1.1.8.

- `OPT_WRITE_TIMEOUT`

  The timeout in seconds for each attempt to write to the server. This option corresponds to the `MYSQL_OPT_WRITE_TIMEOUT` option for the `mysql_options()` C API function. The value is an unsigned integer.

- `password`

  The password for the client MySQL account. This option corresponds to the `passwd` argument of the `mysql_real_connect()` C API function. The value is a string.

- `pipe`

  The name of the named pipe for a named-pipe connection to the local host on Windows systems. The value is a string.

- `pluginDir`

  The directory in which to look for client plugins. This option corresponds to the `MYSQL_PLUGIN_DIR` option for the `mysql_options()` C API function. The value is a string.

  This option was added in Connector/C++ 1.1.5.

- `port`

  The port number for TCP/IP connections. This option corresponds to the `port` argument of the `mysql_real_connect()` C API function. The value is an unsigned integer.

- `postInit`

This option is similar to `preInit`, but the statements are executed after driver initialization. The value is a string.

This option was added in Connector/C++ 1.1.2.

- `preInit`

  A string containing statements to execute before driver initialization. This option corresponds to the `MYSQL_INIT_COMMAND` option for the `mysql_options()` C API function. The value is a string.

  This option was added in Connector/C++ 1.1.2.

- `readDefaultFile`

  Read options from the named option file instead of from `my.cnf`. This option corresponds to the `MYSQL_READ_DEFAULT_FILE` option for the `mysql_options()` C API function. The value is a string.

  This option was added in Connector/C++ 1.1.5.

- `readDefaultGroup`

  Read options from the named group from `my.cnf` or the file specified with `readDefaultFile`. This option corresponds to the `MYSQL_READ_DEFAULT_GROUP` option for the `mysql_options()` C API function. The value is a string.

  This option was added in Connector/C++ 1.1.5.

- `rsaKey`

  The path name to a file containing the server RSA public key. This option corresponds to the `MYSQL_SERVER_PUBLIC_KEY` option for the `mysql_options()` C API function. The value is a string.

- `schema`

  The default database name. This option corresponds to the `db` argument of the `mysql_real_connect()` C API function. The value is a string.

- `socket`

  The name of a Unix domain socket file for a socket-file connection to the local host on Unix and Unix-like systems. This option corresponds to the `socket` argument of the `mysql_real_connect()` C API function. The value is a string.

- `sslCA`

  The path to a file in PEM format that contains a list of trusted SSL CAs. This option corresponds to the `MYSQL_OPT_SSL_CA` option for the `mysql_options()` C API function. The value is a string.

- `sslCAPath`

  The path to a directory that contains trusted SSL CA certificates in PEM format. This option corresponds to the `MYSQL_OPT_SSL_CAPATH` option for the `mysql_options()` C API function. The value is a string.

- `sslCert`

The name of an SSL certificate file in PEM format to use for establishing a secure connection. This option corresponds to the `MYSQL_OPT_SSL_CERT` option for the `mysql_options()` C API function. The value is a string.

- `sslCipher`

  The list of permitted ciphers for SSL encryption. This option corresponds to the `MYSQL_OPT_SSL_CIPHER` option for the `mysql_options()` C API function. The value is a string.

- `sslCRL`

  The path to a file containing certificate revocation lists in PEM format. This option corresponds to the `MYSQL_OPT_SSL_CRL` option for the `mysql_options()` C API function. The value is a string.

  This option was added in Connector/C++ 1.1.4.

- `sslCRLPath`

  The path to a directory that contains files containing certificate revocation lists in PEM format. This option corresponds to the `MYSQL_OPT_SSL_CRLPATH` option for the `mysql_options()` C API function. The value is a string.

  This option was added in Connector/C++ 1.1.4.

- `sslEnforce`

  Whether to require the connection to use SSL. This option corresponds to the `MYSQL_OPT_SSL_ENFORCE` option for the `mysql_options()` C API function. The value is a boolean.

- `sslKey`

  The name of an SSL key file in PEM format to use for establishing a secure connection. This option corresponds to the `MYSQL_OPT_SSL_KEY` option for the `mysql_options()` C API function. The value is a string.

- `sslVerify`

  Enable or disable verification of the server's Common Name value in its certificate against the host name used when connecting to the server. This option corresponds to the `MYSQL_OPT_SSL_VERIFY_SERVER_CERT` option for the `mysql_options()` C API function. The value is a boolean.

  This option was added in Connector/C++ 1.1.4.

- `useLegacyAuth`

  Whether to permit connections to a server that does not support the password hashing used in MySQL 4.1.1 and later. This option corresponds to the `MYSQL_SECURE_AUTH` option for the `mysql_options()` C API function, except that the sense of `useLegacyAuth` is logically opposite that of `MYSQL_SECURE_AUTH`. For example, to disable secure authentication, pass a `useLegacyAuth` value of true. The value is a boolean.

  This option was added in Connector/C++ 1.1.4.

- `userName`

The user name for the MySQL account to use. This option corresponds to the `user` argument of the `mysql_real_connect()` C API function. The value is a string.

Chapter 11 Connector/C++ Known Bugs and Issues

Please report bugs through the MySQL Bug System. See How to Report Bugs or Problems.

For notes detailing the changes in each release of Connector/C++, see MySQL Connector/C++ Release Notes.

- When linking against a static library for 1.0.3 on Windows, 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 cause problems. If you obtain error messages that you suspect are related to binary incompatibilities, build Connector/C++ from source, using the same compiler and linker that you use to build and link your application.

  Due to variations between Linux distributions, compiler versions, linker versions, and STL versions, it is not possible to provide binaries for every possible configuration. However, Connector/C++ binary distribution packages include a `BUILDINFO.txt` file that describes the environment and configuration options used to build the binary versions of the connector libraries. (Prior to Connector/C++ 1.1.11, check the `README` file instead.)

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

Chapter 12 Connector/C++ Support

For general discussion of Connector/C++, please use the C/C++ community forum or join the Connector/C++ mailing list.

Bugs can be reported at the MySQL bug website.

For notes detailing the changes in each release of Connector/C++, see MySQL Connector/C++ Release Notes.

For Licensing questions, and to purchase MySQL Products and Services, please see http://www.mysql.com/buy-mysql/.

Index

E

H

L

M

N

O

P

R

S

T

U