MySQL Connector/C++ 8.0 Developer Guide

Abstract

This manual describes how to install and configure MySQL Connector/C++ 8.0, which provides C++ and plain C interfaces for communicating with MySQL servers, and how to use Connector/C++ to develop database applications.

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.

Document generated on: 2018-06-22 (revision: 57811)

Table of Contents

Preface and Legal Notices

This manual describes how to install and configure MySQL Connector/C++ 8.0, 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++ 8.0 is a MySQL database connector for C++ applications that connect to MySQL servers. Connector/C++ can be used to access MySQL servers that implement a document store, or in a traditional way using SQL queries. It enables development of C++ applications using X DevAPI, or plain C applications using X DevAPI for C.

Connector/C++ 8.0 also enables development of C++ applications that use the legacy JDBC-based API from Connector/C++ 1.1. However, the preferred development environment for Connector/C++ 8.0 is to use X DevAPI or X DevAPI for C.

Connector/C++ applications that use X DevAPI or X DevAPI for C require a MySQL server that has X Plugin enabled. For Connector/C++ applications that use the legacy JDBC-based API, X Plugin is not required or supported.

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 these application programming interfaces:
  - X DevAPI
  - X DevAPI for C
  - 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 X DevAPI

Connector/C++ implements X DevAPI, as described in X DevAPI User Guide. X DevAPI enables connecting to MySQL servers that implement a document store with X Plugin. X DevAPI also enables applications to execute plain SQL queries.

Connector/C++ and X DevAPI for C

Connector/C++ implements a plain C interface called X DevAPI for C that offers functionality similar to that of X DevAPI and that can be used by applications written in plain C. X DevAPI for C enables connecting to MySQL servers that implement a document store with X Plugin. X DevAPI for C also enables applications to execute plain SQL queries.

Connector/C++ and JDBC Compatibility

Connector/C++ implements the JDBC 4.0 API, if built to include the legacy JDBC connector:

- Connector/C++ binary distributions include the JDBC connector.
- If you build Connector/C++ from source, the JDBC connector is not built by default, but can be included by enabling the `WITH_JDBC CMake` option. See Chapter 4, *Installing Connector/C++ from Source*.

The Connector/C++ JDBC API 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.

> **Note**
>
> For more information about the using the Connector/C++ JDBC API, see MySQL Connector/C++ 1.1 Developer Guide.

Platform Support and Prerequisites

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

Commercial and Community Connector/C++ distributions require the Visual C++ Redistributable for Visual Studio 2015 to work on Windows platforms. 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, the MySQL server requirements depend on the API the application uses:
  - Applications that use the JDBC API can use a server from MySQL 5.5 or higher.
  - Connector/C++ applications that use X DevAPI or X DevAPI for C require a server from MySQL 8.0 (8.0.11 or higher) or MySQL 5.7 (5.7.12 or higher), with X Plugin enabled. For MySQL 8.0, X Plugin is enabled by default. For MySQL 5.7, it must be enabled explicitly. (Some X Protocol features may not work with MySQL 5.7.)

  In addition, applications that use MySQL features available only in MySQL 8.0 or higher require a server from MySQL 8.0 or higher.

- To build Connector/C++ from source:
  - The MySQL C API client library may be required:
    - Building the JDBC connector requires a client library from MySQL 5.7 (5.7.9 or higher) or MySQL 8.0 (8.0.11 or higher). This occurs when Connector/C++ is configured with the `WITH_JDBC` `CMake` option enabled to include the JDBC connector.
    - For Connector/C++ built without the JDBC connector, the client library is not needed.
  - 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**
>
> Commercial and Community Connector/C++ distributions require the Visual C ++ Redistributable for Visual Studio 2015 to work on Windows platforms. 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.** For Connector/C++ 8.0, the MSI Installer is available as of Connector/C++ 8.0.12. To use the MSI Installer (`.msi` file), launch it and follow the prompts in the screens it presents. The MSI Installer can install components for two connectors:

  - The connector for X DevAPI (including X DevAPI for C).

  - The connector for the legacy JDBC API.

  For each connector, there are two components:

  - The DLL component includes the connector DLLs and libraries to satisfy runtime dependencies. This component is required to run Connector/C++ application binaries that use the connector.

  - The Developer component includes header files, static libraries, and import libraries for DLLs. This component is required to build from source Connector/C++ applications that use the connector.

  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 **Complete** installation installs both components for both connectors.

  - The **Typical** installation installs the DLL component for both connectors.

  - The **Custom** installation enables you to select which components to install. Both components for the X DevAPI connector are preselected, but you can override the selection. The Developer component for a connector cannot be selected without also selecting the connector DLL component.

    The **Custom** installation also enables you to specify the installation location.

  For all installation types, the MSI Installer performs these actions:

  - It checks whether the required Visual C++ Redistributable for Visual Studio 2015 is present. If not, the installer asks you to install it and exits with an error.

  - It installs documentation files: `README.txt`, `LICENSE.txt`, `BUILDINFO.txt`.

- **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.

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
- SSL Support

Build Tools

You must have the cross-platform build tool `CMake` (2.8.12 or higher).

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

MySQL Client Library

To build Connector/C++ from source, the MySQL C API client library may be required:

- Building the JDBC connector requires a client library from MySQL 5.7 (5.7.9 or higher) or MySQL 8.0 (8.0.11 or higher). This occurs when Connector/C++ is configured with the `WITH_JDBC` `CMake` option enabled to include the JDBC connector.
- For Connector/C++ built without the JDBC connector, the client library is not needed.

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 4.4, “Connector/C++ Source-Configuration Options”).

Boost C++ Libraries

To compile Connector/C++ the Boost C++ libraries are needed only if you build the legacy JDBC API or if the version of the C++ standard library on your system does not implement the UTF8 converter (`codecvt_utf8`).

If the Boost C++ libraries are 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 `WITH_BOOST` `CMake` option to indicate where the Boost files are located (see Section 4.4, “Connector/C++ Source-Configuration Options”):

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

Adjust the path as necessary to match your installation.

SSL Support

Use the `WITH_SSL` `CMake` option to specify which SSL library to use when compiling Connector/C++. These SSL library choices are available:

- As of Connector/C++ 8.0.12: Connector/C++ can be compiled using OpenSSL or wolfSSL. The default is OpenSSL.
- Prior to Connector/C++ 8.0.12: Connector/C++ can be compiled using OpenSSL or yaSSL. The default is yaSSL.

If you compile Connector/C++ using OpenSSL, OpenSSL 1.0.x is required.

To compile Connector/C++ using wolfSSL, you must download the wolfSSL source code to your system. To obtain it, visit https://www.wolfssl.com. Connector/C++ requires wolfSSL 3.14.0 or higher. Also, because wolfSSL uses TLSv1.2 by default, SSL connections from Connector/C++ applications to MySQL servers that do not support TLSv1.2 are not supported. For example, MySQL 5.7 Community distributions are compiled using yaSSL, which does not support TLSv1.2. This means that MySQL 5.7 Community servers cannot be used with Connector/C++ built using wolfSSL.

For more information about `WITH_SSL` and SSL libraries, see Section 4.4, “Connector/C++ Source-Configuration Options”.

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` branche (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
```

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

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

- Configuring Connector/C++
- Building Connector/C++
- Installing Connector/C++
- Verifying Connector/C++ Functionality

Configuring Connector/C++

Use `CMake` to configure and build Connector/C++. Only out-of-source-builds are supported, so create a directory to use for the build and change location into it. Then configure the build using this command, where *`concpp_source`* is the directory containing the Connector/C++ source code:

```
cmake concpp_source
```

It may be necessary to specify other options on the configuration command. Some examples:

- By default, these installation locations are used:
  - `/usr/local/mysql/connector-c++-8.0` (Unix and Unix-like systems)
  - *`User_home`*`/MySQL/"MySQL Connector C++ 8.0"` (Windows)

  To specify the installation location explicitly, use the `CMAKE_INSTALL_PREFIX` option:

  ```
  -DCMAKE_INSTALL_PREFIX=path_name
  ```

- On Windows, you can use the `-G` option to select a particular generator:
  - `-G "Visual Studio 14 2015 Win64"` (64-bit builds)
  - `-G "Visual Studio 14 2015"` (32-bit builds)

  Consult the `CMake` manual or check `cmake --help` to find out which generators are supported by your `CMake` version. (However, it may be that your version of `CMake` supports more generators than can actually be used to build Connector/C++.)

- If the Boost C++ libraries are needed, use the `WITH_BOOST` option to specify their location:

```
-DWITH_BOOST=path_name
```

- By default, the build creates dynamic (shared) libraries. To build static libraries, enable the `BUILD_STATIC` option:

```
-DBUILD_STATIC=ON
```

- By default, the legacy JDBC connector is not built. If you plan to build this connector, an additional `git` command is needed to perform submodule initialization (do this in the top-level source directory):

```
git submodule update --init
```

  To include the JDBC connector in the build, enable the `WITH_JDBC` option:

```
-DWITH_JDBC=ON
```

> **Note**
>
> If you configure and build the test programs later, use the same `CMake` options to configure them as the ones you use to configure Connector/C++ (`-G`, `WITH_BOOST`, `BUILD_STATIC`, and so forth). Exceptions: Path name arguments will differ, and you need not specify `CMAKE_INSTALL_PREFIX`.

For information about `CMake` configuration options, see Section 4.4, “Connector/C++ Source-Configuration Options”.

Building Connector/C++

After configuring the Connector/C++ distribution, build it using this command:

```
cmake --build . --config build_type
```

The `--config` option is optional. It specifies the build configuration to use, such as `Release` or `Debug`. If you omit `--config`, the default is `Debug`.

> **Important**
>
> If you specify the `--config` option on the preceding command, specify the same `--config` option for later steps, such as the steps that install Connector/C++ or that build test programs.

If the build is successful, it creates the connector libraries in the build directory. (For Windows, look for the libraries in a subdirectory with the same name as the *`build_type`* value specified for the `--config` option.)

- If you build dynamic libraries, they have these names:
  - `libmysqlcppconn8.so.1` (Unix)
  - `libmysqlcppconn8.1.dylib` (macOS)
  - `mysqlcppconn8-1-vs14.dll` (Windows)

- If you build static libraries, they have these names:
  - `libmysqlcppconn8-static.a` (Unix, macOS)
  - `mysqlcppconn8-static.lib` (Windows)

If you enabled the `WITH_JDBC` option to include the legacy JDBC connector in the build, the following additional library files are created.

- If you build legacy dynamic libraries, they have these names:
  - `libmysqlcppconn.so.7` (Unix)
  - `libmysqlcppconn.7.dylib` (macOS)
  - `mysqlcppconn-7-vs14.dll` (Windows)
- If you build legacy static libraries, they have these names:
  - `libmysqlcppconn-static.a` (Unix, macOS)
  - `mysqlcppconn-static.lib` (Windows)

Installing Connector/C++

To install Connector/C++, use this command:

```
cmake --build . --target install --config build_type
```

Verifying Connector/C++ Functionality

To verify connector functionality, build and run one or more of the test programs included in the `testapp` directory of the source distribution. Create a directory to use and change location into it. Then issue the following commands:

```
cmake [other_options] -DWITH_CONCPP=concpp_install concpp_source/testapp
cmake --build . --config=build_type
```

`other_options` consists of the options that you used to configure Connector/C++ itself (`-G`, `WITH_BOOST`, `BUILD_STATIC`, and so forth). *`concpp_source`* is the directory containing the Connector/C++ source code, and *`concpp_install`* is the directory where Connector/C++ is installed:

The preceding commands should create the `devapi_test` and `xapi_test` programs in the `run` directory of the build location. If you enable `WITH_JDBC` when configuring the test programs, the build also creates the `jdbc_test` program.

Before running test programs, ensure that a MySQL server instance is running with X Plugin enabled. The easiest way to arrange this is to use the `mysql-test-run.pl` script from the MySQL distribution. For MySQL 8.0, X Plugin is enabled by default, so invoke this command in the `mysql-test` directory of that distribution:

```
perl mysql-test-run.pl --start-and-exit
```

For MySQL 5.7, X Plugin must be enabled explicitly, so add an option to do that:

```
perl mysql-test-run.pl --start-and-exit --mysqld=--plugin-load=mysqlx
```

The command should start a test server instance with X Plugin enabled and listening on port 13009 instead of its standard port (33060).

Now you can run one of the test programs. They accept a connection-string argument, so if the server was started as just described, you can run them like this:

```
run/devapi_test mysqlx://root@127.0.0.1:13009
run/xapi_test mysqlx://root@127.0.0.1:13009
```

The connection string assumes availability of a `root` user account without any password and the programs assume that there is a `test` schema available (assumptions that hold for a server started using `mysql-test-run.pl`).

To test `jdbc_test`, you need a MySQL server, but X Plugin is not required. Also, the connection options must be in the form specified by the JDBC API. Pass the user name as the second argument. For example:

```
run/jdbc_test tcp://127.0.0.1:13009 root
```

4.4 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
`BUILD_STATIC`	Whether to build a static librarty	`OFF`	
`BUNDLE_DEPENDENCIES`	Whether to bundle external dependency libraries with the connector	`OFF`	
`CMAKE_BUILD_TYPE`	Type of build to produce	`RelWithDebInfo`	
`CMAKE_INSTALL_PREFIX`	Installation base directory	`/usr/local`	
`MAINTAINER_MODE`	For internal use only	`OFF`	8.0.12
`MYSQL_CONFIG_EXECUTABLE`	Path to the mysql_config program	`${MYSQL_DIR}/bin/mysql_config`	
`MYSQL_DIR`	MySQL Server or Connector/C installation directory		
`STATIC_MSVCRT`	Use the static runtime library		
`WITH_BOOST`	The Boost source directory		
`WITH_CONCPP`	The location where Connector/C++ is installed		
`WITH_JDBC`	Whether to build legacy JDBC library	`OFF`	8.0.7
`WITH_SSL`	Type of SSL support	`bundled`	8.0.7

- `-DBUILD_STATIC=`*`bool`*

  By default, dynamic (shared) libraries are built. If this option is enabled, static libraries are built instead.

- `-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.
  - `Release`: Enable optimizations.
  - `RelWithDebInfo`: Enable optimizations and generate debugging information.

- `-DCMAKE_INSTALL_PREFIX=dir_name`

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

- `-DMAINTAINER_MODE=bool`

  This is an internal option used for creating Connector/C++ distribution packages. It was added in Connector/C++ 8.0.12.

- `-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.

  This option applies only if you are building the legacy JDBC connector (that is, only if `WITH_JDBC` is enabled).

- `-DMYSQL_DIR=dir_name`

  The directory where MySQL is installed.

  This option applies only if you are building the legacy JDBC connector (that is, only if `WITH_JDBC` is enabled).

- `-DSTATIC_MSVCRT=bool`

  (Windows only) Use the static runtime library (the `/MT*` compiler option). This option might be necessary if code that uses Connector/C++ also uses the static runtime library.

- `-DWITH_BOOST=dir_name`

  The directory where the Boost sources are installed.

- `-DWITH_CONCPP=path_name`

  The location where Connector/C++ is installed. This is used when configuring the test programs so the build can find the Connector/C++ header files and libraries.

- `-DWITH_JDBC=bool`

  Whether to build the legacy JDBC connector. This option is disabled by default. If it is enabled, Connector/C++ 8.0 applications can use the legacy JDBC API, just like Connector/C++ 1.1 applications.

- `-DWITH_SSL={ssl_type|path_name}`

  This option specifies which SSL library to use when compiling Connector/C++. These SSL library choices are available:

- As of Connector/C++ 8.0.12: Connector/C++ can be compiled using OpenSSL or wolfSSL. The default is OpenSSL. wolfSSL (3.14.0 or higher) can be used if the wolfSSL sources are present on your system.
- Prior to Connector/C++ 8.0.12: Connector/C++ can be compiled using OpenSSL or yaSSL. The default is yaSSL, which is bundled with Connector/C++.

The `WITH_SSL` value indicates the type of SSL support to include or the path name to the SSL installation to use:

- *`ssl_type`* can be one of the following values:
  - `system`: Use the system OpenSSL library. This is the default as of Connector/C++ 8.0.12.

    When running an application that is linked to the connector dynamic library, the OpenSSL libraries on which the connector depends should be correctly found if they are placed in the file system next to the connector library. The application should also work when the OpenSSL libraries are installed at the standard system-wide locations. This assumes that the version of OpenSSL is as expected by Connector/C++.

    Compressed `tar` files or Zip archive distributions for for Windows, Linux, and macOS should contain the required OpenSSL libraries in the same location as the connector library.

    Except for Windows, it should be possible to run an application linked to the connector dynamic library when the connector library and the OpenSSL libraries are placed in a nonstandard location, provided that these locations were stored as runtime paths when building the application (`gcc -rpath` option).

    For Windows, an application that is linked to the connector shared library can be run only if the connector library and the OpenSSL libraries are stored either:

    - In the Windows system folder
    - In the same folder as the application
    - In a folder listed in the `PATH` environment variable

    If the application is linked to the connector static library, it remains true that the required OpenSSL libraries must be found in one of the preceding locations.
  - `bundled`: Use the yaSSL library bundled with the distribution rather than OpenSSL. This option value is available only prior to Connector/C++ 8.0.12 (and is the default prior to 8.0.12). As of Connector/C++ 8.0.12, wolfSSL is the alternative to OpenSSL, so yaSSL is no longer bundled and this option value is neither valid nor the default.
- *`path_name`* is the path name to the SSL installation to use:
  - As of Connector/C++ 8.0.12: *`path_name`* can be the path to the installed OpenSSL library or the wolfSSL sources.
  - Prior to Connector/C++ 8.0.12: *`path_name`* must be the path to the installed OpenSSL library.

  For OpenSSL, the path must point to a directory containing a `lib` subdirectory with OpenSSL libraries that are already built. For wolfSSL, the path must point to the source directory where the wolfSSL sources are located.

Specifying a path name for the OpenSSL installation can be preferable to using the *`ssl_type`* value of `system` because it can prevent `CMake` from detecting and using an older or incorrect OpenSSL version installed on the system.

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++ using command-line tools. This section applies to any platform that supports the tools. See Section 5.2, “Building Connector/C++ 8.0 Applications from the Command Line with make”.
- Building applications for Connector/C++ on Windows using Visual Studio. See Section 5.3, “Building Connector/C++ Applications on Windows with Microsoft Visual Studio”.

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

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

- Build Tools and Configuration Settings
- C++11 Support
- Boost Header Files
- Link Libraries
- 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 the 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 distributions 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.

C++11 Support

X DevAPI uses C++11 language features. For that reason, enable C++11 support in the compiler using the `-std=c++11` option when building Connector/C++ applications that use X DevAPI. This option is not needed for applications that use X DevAPI for C (which is a plain C API) or the legacy JDBC API (which is based on plain C++), unless the application code uses C++11.

Boost Header Files

To compile applications that use Connector/C++, the Boost header files are needed if you use the legacy JDBC API. For applications that use X DevAPI or X DevAPI for C, the Boost header files are needed on Unix and Unix-like platforms if you build using `gcc` and the version of the C++ standard library on your system does not implement the UTF8 converter (`codecvt_utf8`).

If the Boost header files are 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.

Link Libraries

When Connector/C++ is built using OpenSSL, it makes the connector library dependent on OpenSSL dynamic libraries:

- When linking an application to Connector/C++ dynamically, this dependency is relevant only at runtime.
- When linking an application to Connector/C++ statically, the OpenSSL libraries should be linked as well. On Linux, this means adding `-lssl -lcrypto` explicitly to the compile/link command. On Windows, this is handled automatically.

On Windows, link to the dynamic version of the C++ Runtime Library.

Runtime Libraries

X DevAPI for C applications need `libstdc++` at runtime. Depending on your platform or build tools, a different library may apply. For example, the library is `libc++` on macOS; see macOS Notes.

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.

Connector/C++ libraries built by Oracle depend on the OpenSSL libraries. The latter must be installed on the system in order to run code that links against Connector/C++ libraries. Another option is to put the OpenSSL libraries in the same location as Connector/C++ In this case the dynamic linker should find them next to the connector library. See also macOS Notes, and Windows Notes.

It is possible to build Connector/C++ with an SSL library other than OpenSSL. Use the `WITH_SSL` `CMake` option to specify which SSL library to use (see Section 4.4, “Connector/C++ Source-Configuration Options”). For applications linked against Connector/C++ libraries that are not built using OpenSSL, the OpenSSL libraries are not required at runtime.

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++ 8.0 Applications from the Command Line with make

This section discusses building Connector/C++ 8.0 applications from the command line using `make`. It applies to any platform that supports `make` and command-line build tools such as `g++`, `cc`, or `clang`. For information about building Connector/C++ 8.0 applications on Windows using a graphical interface, see Section 5.3, “Building Connector/C++ Applications on Windows with Microsoft Visual Studio”.

- Connector/C++ Header Files
- Using the Connector/C++ Dynamic Library
- Using the Connector/C++ Static Library
- macOS Notes
- Windows Notes

Connector/C++ Header Files

The API an application uses determines which Connector/C++ header files it should include. The following include directives work under the assumption that the include path contains `$MYSQL_CPPCONN_DIR/include`, where `$MYSQL_CPPCONN_DIR` is the Connector/C++ installation location. Pass an `-I $MYSQL_CPPCONN_DIR/include` option on the compiler invocation command to ensure this.

- For applications that use X DevAPI:

```
#include <mysqlx/xdevapi.h>
```

- For applications that use X DevAPI for C:

```
#include <mysqlx/xapi.h>
```

- For applications that use the legacy JDBC API:

```
#include <jdbc/mysql_driver.h>
#include <jdbc/mysql_connection.h>
#include <jdbc/cppconn/*.h>
```

Legacy code that uses Connector/C++ 1.1 has `#include` directives of this form:

```
#include <mysql_driver.h>
#include <mysql_connection.h>
#include <cppconn/*.h>
```

To build such code with Connector/C++ 8.0 without modifying it, add `$MYSQL_CPPCONN_DIR/include/jdbc` to the include path.

To compile code that you intend to link statically against Connector/C++, define a macro that adjusts API declarations in the header files for usage with the static library. For details, see Using the Connector/C++ Static Library.

Using the Connector/C++ Dynamic Library

The Connector/C++ dynamic library name depends on the platform. These libraries implement X DevAPI and X DevAPI for C:

- `libmysqlcppconn8.so` (Unix)
- `libmysqlcppconn8.1.dylib` (macOS)
- `mysqlcppconn8-1-vs`*`NN`*`.dll`, with import library `vs`*`NN`*`/mysqlcppconn8.lib` (Windows)

For the legacy JDBC API, the dynamic libraries are named as follows:

- `libmysqlcppconn.so.7` (Unix)
- `libmysqlcppconn.7.dylib` (macOS)
- `mysqlcppconn-7-vs`*`NN`*`.dll`, with import library `vs`*`NN`*`/mysqlcppconn-static.lib` (Windows)

On Windows, the `vs`*`NN`* value in library names depends on the MSVC compiler version used to build the libraries (for example, `vs14` for MSVC 2015). This convention enables using libraries built with different versions of MSVC on the same system. For details, see Windows Notes.

To build code that uses X DevAPI or X DevAPI for C, add `-lmysqlcppconn8` to the linker options. To build code that uses the legacy JDBC API, add `-lmysqlcppconn`. You must also indicate whether to use the 64-bit or 32-bit libraries by specifying the appropriate library directory. Use an `-L` linker option to specify `$MYSQL_CONCPP_DIR/lib64` (64-bit libraries) or `$MYSQL_CONCPP_DIR/lib` (32-bit libraries), where `$MYSQL_CPPCONN_DIR` is the Connector/C++ installation location.

To build a Connector/C++ application that uses X DevAPI, has sources in `app.cc`, and links dynamically to the connector library, the `Makefile` might look like this:

```
MYSQL_CONCPP_DIR= Connector/C++ installation location
CPPFLAGS = -I $(MYSQL_CONCPP_DIR)/include -L $(MYSQL_CONCPP_DIR)/lib64
LDLIBS = -lmysqlcppconn8
CXXFLAGS = -std=c++11
app : app.cc
```

With that `Makefile`, the command `make app` generates the following compiler invocation:

```
g++ -std=c++11 -I .../include -L .../lib64 app.cc -lmysqlcppconn8 -o app
```

To build a plain C application that uses X DevAPI for C, has sources in `app.c`, and links dynamically to the connector library, the `Makefile` might look like this:

```
MYSQL_CONCPP_DIR= Connector/C++ installation location
CPPFLAGS = -I $(MYSQL_CONCPP_DIR)/include -L $(MYSQL_CONCPP_DIR)/lib64
LDLIBS = -lmysqlcppconn8
app : app.c
```

With that `Makefile`, the command `make app` generates the following compiler invocation:

```
```

```
cc -I .../include -L .../lib64 app.c -lmysqlcppconn8 -o app
```

Typically, X DevAPI for C applications are written in plain C. However, if application code does use C++11, the `Makefile` should also specify the `-std=c++11` option:

```
CXXFLAGS = -std=c++11
```

> **Note**
>
> The resulting code, even though it is compiled as plain C, depends on the C++ runtime (typically `libstdc++`, though this may differ depending on platform or build tools; see Runtime Libraries).

To build a plain C++ application that uses the legacy JDBC API, has sources in `app.c`, and links dynamically to the connector library, the `Makefile` might look like this:

```
MYSQL_CONCPP_DIR= Connector/C++ installation location
CPPFLAGS = -I $(MYSQL_CONCPP_DIR)/include -L $(MYSQL_CONCPP_DIR)/lib64
LDLIBS = -lmysqlcppconn
app : app.c
```

The library option in this case is `-lmysqlcppcon`, rather than `-lmysqlcppcon8` as for an X DevAPI or X DevAPI for C application.

With that `Makefile`, the command `make app` generates the following compiler invocation:

```
cc -I .../include -L .../lib64 app.c -lmysqlcppconn -o app
```

Typically, applications that use the legacy JDBC API are written in plain C++. However, if application code does use C++11, the `Makefile` should also specify the `-std=c++11` option:

```
CXXFLAGS = -std=c++11
```

> **Note**
>
> When running an application that uses the Connector/C++ dynamic library, the library and its runtime dependencies must be found by the dynamic linker. See Runtime Libraries.

Using the Connector/C++ Static Library

It is possible to link your application with the Connector/C++ static library. This way there is no runtime dependency on the connector, and the resulting binary can run on systems where Connector/C++ is not installed.

> **Note**
>
> Even when linking statically, the resulting code still depends on all runtime dependencies of the Connector/C++ library. For example, if Connector/C++ is built using OpenSSL, the code has a runtime dependency on the OpenSSL libraries. See Runtime Libraries.

The Connector/C++ static library name depends on the platform. These libraries implement X DevAPI and X DevAPI for C:

- `libmysqlcppconn8-static.a` (Unix, macOS)
- `vs*NN*/mysqlcppconn8-static.lib` (Windows)

For the legacy JDBC API, the static libraries are named as follows:

- `libmysqlcppconn-static.a` (Unix, macOS)
- `vs*NN*/mysqlcppconn-static.lib` (Windows)

On Windows, the `vs*NN*` value in library names depends on the MSVC compiler version used to build the libraries (for example, `vs14` for MSVC 2015). This convention enables using libraries built with different versions of MSVC on the same system. For details, see Windows Notes.

To compile code that you intend to link statically against Connector/C++, define a macro that adjusts API declarations in the header files for usage with the static library. One way to define the macro is by passing a `-D` option on the compiler invocation command:

- For applications that use X DevAPI or X DevAPI for C, define the `STATIC_CONCPP` macro. All that matters is that you define it; the value does not matter. For example: `-DSTATIC_CONCPP`
- For applications that use the legacy JDBC API, 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`. For example: `-DCPPCONN_PUBLIC_FUNC=`

To build a Connector/C++ application that uses X DevAPI, has sources in `app.cc`, and links statically to the connector library, the `Makefile` might look like this:

```
MYSQL_CONCPP_DIR= Connector/C++ installation location
CPPFLAGS = -DSTATIC_CONCPP -I $(MYSQL_CONCPP_DIR)/include
LDLIBS = $(MYSQL_CONCPP_DIR)/lib64/libmysqlcppconn8-static.a -lssl -lcrypto -lpthread
CXXFLAGS = -std=c++11
app : app.cc
```

With that `Makefile`, the command `make app` generates the following compiler invocation:

```
g++ -std=c++11 -DSTATIC_CONCPP -I .../include app.cc
  .../lib64/libmysqlcppconn8-static.a -lssl -lcrypto -lpthread -o app
```

> **Note**
>
> To avoid having the linker report unresolved symbols, the compile line must include the OpenSSL libraries and the `pthread` library on which Connector/C++ code depends.
>
> OpenSSL libraries are not needed if Connector/C++ is built without them, but Connector/C++ as built and distributed by Oracle does depend on OpenSSL.
>
> The exact list of libraries required by Connector/C++ library depends on the platform. For example, on Solaris, the `socket`, `rt`, and `nsl` libraries might be needed.

To build a plain C application that uses X DevAPI for C, has sources in `app.c`, and links statically to the connector library, the `Makefile` might look like this:

```
MYSQL_CONCPP_DIR= Connector/C++ installation location
```

```
CPPFLAGS = -DSTATIC_CONCPP -I $(MYSQL_CONCPP_DIR)/include
LDLIBS = $(MYSQL_CONCPP_DIR)/lib64/libmysqlcppconn8-static.a -lssl -lcrypto -lpthread
app : app.c
```

With that `Makefile`, the command `make app` generates the following compiler invocation:

```
cc -DSTATIC_CONCPP -I .../include app.c
  .../lib64/libmysqlcppconn8-static.a -lssl -lcrypto -lpthread -o app
```

If the application code uses C++11, the `Makefile` should also specify the `-std=c++11` option:

```
CXXFLAGS = -std=c++11
```

To build a plain C application that uses the legacy JDBC API, has sources in `app.c`, and links statically to the connector library, the `Makefile` might look like this:

```
MYSQL_CONCPP_DIR= Connector/C++ installation location
CPPFLAGS = -DCPPCONN_PUBLIC_FUNC= -I $(MYSQL_CONCPP_DIR)/include
LDLIBS = $(MYSQL_CONCPP_DIR)/lib64/libmysqlcppconn-static.a -lssl -lcrypto -lpthread
app : app.c
```

The library option in this case names `libmysqlcppcon-static.a`, rather than `libmysqlcppcon8-static.a` as for an X DevAPI or X DevAPI for C application.

With that `Makefile`, the command `make app` generates the following compiler invocation:

```
cc -std=c++11 --DCPPCONN_PUBLIC_FUNC= -I .../include app.c
  .../lib64/libmysqlcppconn-static.a -lssl -lcrypto -lpthread -o app
```

If the application code uses C++11, the `Makefile` should also specify the `-std=c++11` option:

```
CXXFLAGS = -std=c++11
```

When building plain C code, it is important to take care of connector's dependency on the C++ runtime, which is introduced by the connector library even though the code that uses it is plain C:

- One approach is to ensure that a C++ linker is used to build the final code. This approach is taken by the `Makefile` shown here:

  ```
  MYSQL_CONCPP_DIR= Connector/C++ installation location
  CPPFLAGS = -DSTATIC_CONCPP -I $(MYSQL_CONCPP_DIR)/include
  LDLIBS = $(MYSQL_CONCPP_DIR)/lib64/libmysqlcppconn8-static.a -lssl -lcrypto -lpthread
  LINK.o = $(LINK.cc) # use C++ linker
  app : app.o
  ```

  With that `Makefile`, the build process has two steps: first the application source in `app.c` is compiled using a plain C compiler to produce `app.o`, then the final executable (`app`) is linked using the C++ linker, which takes care of the dependency on the C++ runtime:

  ```
  cc -DSTATIC_CONCPP -I .../include -c -o app.o app.c
  g++ -DSTATIC_CONCPP -I .../include app.o
    .../libmysqlcppconn8-static.a -lssl -lcrypto -lpthread -o app
  ```

- Another approach is to use a plain C compiler and linker, but add the `libstdc++` C++ runtime library as an explicit option to the linker. This approach is taken by the `Makefile` shown here:

```
MYSQL_CONCPP_DIR= Connector/C++ installation location
CPPFLAGS = -DSTATIC_CONCPP -I $(MYSQL_CONCPP_DIR)/include
LDLIBS = $(MYSQL_CONCPP_DIR)/lib64/libmysqlcppconn8-static.a -lssl -lcrypto -lpthread -lstdc++
app : app.c
```

With that `Makefile`, the compiler is invoked as follows:

```
cc -DSTATIC_CONCPP -I .../include app.c
  .../libmysqlcppconn8-static.a -lssl -lcrypto -lpthread -lstdc++ -o app
```

> **Note**
>
> Even if the application that uses Connector/C++ is written in plain C, the final executable depends on the C++ runtime which must be installed on the target computer on which the application will run.

macOS Notes

The binary distribution of Connector/C++ for macOS is compiled using the macOS native `clang` compiler. For that reason, an application that uses Connector/C++ should be built with the same `clang` compiler.

The `clang` compiler can use two different implementations of the C++ runtime library: either the native `libc++` or the GNU `libstdc++` library. It is important that an application uses the same runtime implementation as Connector/C++ that is, the native `libc++`. To ensure that, the `-stdlib=libc++` option should be passed to the compiler and the linker invocations.

To build a Connector/C++ application that uses X DevAPI, has sources in `app.cc`, and links dynamically to the connector library, the `Makefile` for building on macOS might look like this:

```
MYSQL_CONCPP_DIR= Connector/C++ installation location
CPPFLAGS = -I $(MYSQL_CONCPP_DIR)/include -L $(MYSQL_CONCPP_DIR)/lib64
LDLIBS = -lmysqlcppconn8
CXX = clang++ -stdlib=libc++
CXXFLAGS = -std=c++11
app : app.cc
```

Binary packages for macOS include OpenSSL libraries that are required by code linked with the connector. These libraries are installed in the same location as the connector libraries and should be found there by the dynamic linker.

Windows Notes

On Windows, applications can be built in different modes (also called build configurations), which determine the type of the runtime library that is used by the final executable. An application can be built in debug or release mode. Then it can be built in 32-bit or 64-bit mode. Also, one can choose between the static (`/MT`) or the dynamic (`/MD`) runtime. Different versions of the MSVC compiler also use different versions of the runtime.

It is important to ensure that the compiler version and the build mode of an application match the same parameters used when building the connector library, to ensure that the connector and the application use the same runtime library.

The binary distribution of Connector/C++ 8.0 ships libraries built in release mode using dynamic runtime (`/MD`). The libraries are built with MSVC 2015 (the exact compiler version can be found in the

`BUILDINFO.txt` file included in the package). Consequently, the code that uses these libraries must be built with the same version of MSVC and in `/MD` mode. To build code in a different mode, first build Connector/C++ from source in that mode (see Section 4.3, "Installing Connector/C++ from Source"), then build your applications using the same mode.

> **Note**
>
> 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++ in debug mode first.

There are separate 64-bit and 32-bit packages, each keeping libraries in the `lib64` and `lib` directories, respectively. Package and library names also include `vs`*`NN`*. The `vs`*`NN`* value in these names depends on the MSVC compiler version used to build the libraries (for example, `vs14` for MSVC 2015). This convention enables using libraries built with different versions of MSVC on the same system.

A dynamic connector library name has a `.dll` extension and is used with a import library that has a `.lib` extension in the `vs`*`NN`* subdirectory. Thus, a connector dynamic library named `mysqlcppconn8-1-vs14.dll` is used with the import library named `vs14/mysqlcppconn8.lib`. The 1 in the dynamic library name is the major ABI version number. (In the future, this will help using compatibility libraries with old ABI together with new libraries having a different ABI.) The `vs14` subdirectory also contains a static version of the library named `vs14/mysqlcppconn8-static.lib`.

A legacy JDBC connector dynamic library named `mysqlcppconn-7-vs14.dll` is used with the import library named `vs14/mysqlcppconn.lib`. The corresponding static library is named `vs14/mysqlcppconn-static.lib`.

When building code that uses Connector/C++ libraries:

- Add `$MYSQL_CPPCONN_DIR/include` as an additional include directory and `$MYSQL_CONCPP_DIR/lib64/vs14` (64-bit libraries) or `$MYSQL_CONCPP_DIR/lib/vs14` (32-bit libraries) as an additional library directory in the project configuration.
- To use the dynamic library, add `mysqlcppconn8.lib` to the linker options (or `mysqlcppconn.lib` for legacy code).
- For use the static library, add `mysqlcppconn8-static.lib` (or `mysqlcppconn-static.lib` for legacy code).

If linking statically, the linker must find the link libraries (with `.lib` extension) for the required OpenSSL libraries. If the connector was installed from the binary package provided by Oracle, they are present in the `$MYSQL_CONCPP_DIR/lib64/vs14` or `$MYSQL_CONCPP_DIR/lib/vs14` directory, and the corresponding OpenSSL `.dll` libraries are present in the main library directory, next to the connector `.dll` libraries.

> **Note**
>
> An application that uses the connector dynamic library must be able to locate it at runtime, as well as its dependencies such as OpenSSL. The common way of arranging this is to put the required DLLs in the same location as the executable.

5.3 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. the name of the last path component differs depending on whether you use a 64-bit or 32-bit package. The name ends with `lib64/vs14` for 64-bit packages, `lib/vs14` for 32-bit packages. In addition, import library files are found in a subdirectory named for the build type used to build the libraries, here designated as `vs14`.

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

Connector Type	Import Library	Dynamic Library
X DevAPI, X DevAPI for C	*`LIB`*`/vs14/mysqlcppconn8.lib`	*`LIB`*`/mysqlcppconn8-1-vs14.dll`
JDBC	*`LIB`*`/vs14/mysqlcppconn.lib`	*`LIB`*`/mysqlcppconn-7-vs14.dll`

Linking the Static Library

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

The following table indicates which static library files to use for Connector/C++. *`LIB`* denotes the Connector/C++ installation library path name. The name of the last path component differs depending on whether you use a 64-bit or 32-bit package. The name ends with `lib64` for 64-bit packages, `lib` for 32-bit packages. In addition, static library files are found in a subdirectory named for the build type used to build the libraries, here designated as `vs14`.

Table 5.2 Connector/C++ Static Library Names

Connector Type	Static Library
X DevAPI, X DevAPI for C	*`LIB`*`/vs14/mysqlcppconn8-static.lib`
JDBC	*`LIB`*`/vs14/mysqlcppconn-static.lib`

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**.

   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.

   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:

   - For Connector/C++ applications that use X DevAPI or X DevAPI for C, define the `STATIC_CONCPP` macro. All that matters is that you define it; the value does not matter.
   - For Connector/C++ applications that use the legacy JDBC API, 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**.

   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.

   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.

Chapter 6 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 distributions include a `BUILDINFO.txt` file that describes the environment and configuration options used to build the binary versions of the connector libraries.
- 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 the release build of Connector/C++ with a debug build of the client application.

Chapter 7 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