# MySQL++ v3.2.5 User Manual

Kevin Atkinson
Sinisa Milivojevic
Monty Widenius
Warren Young



February 27, 2020

## Table of Contents

# 1. Introduction

MySQL++ is a powerful C++ wrapper for MySQL's C API[1]. Its purpose is to make working with queries as easy as working with STL containers.

The latest version of MySQL++ can be found at the official web site.

Support for MySQL++ can be had on the mailing list. That page hosts the mailing list archives, and tells you how you can subscribe.

## 1.1. A Brief History of MySQL++

MySQL++ was created in 1998 by Kevin Atkinson. It started out MySQL-specific, but there were early efforts to try and make it database-independent, and call it SQL++. This is where the old library name "sqlplus" came from. This is also why the old versions prefixed some class names with "Mysql" but not others: the others were supposed to be the database-independent parts. All of Kevin's releases had pre-1.0 version numbers.

Then in 1999, MySQL AB took over development of the library. In the beginning, Monty Widenius himself did some of the work, but later gave it over to another MySQL employee, Sinisa Milivojevic. MySQL released versions 1.0 and 1.1, and then Kevin gave over maintenance to Sinisa officially with 1.2, and ceased to have any involvement with the library's maintenance. Sinisa went on to maintain the library through 1.7.9, released in mid-2001. It seems to be during this time that the dream of multiple-database compatibility died, for obvious reasons.

With version 1.7.9, MySQL++ went into a period of stasis, lasting over three years. (Perhaps it was the ennui and retrenchment following the collapse of the bubble that caused them to lose interest.) During this time, Sinisa ran the MySQL++ mailing list and supported its users, but made no new releases. Contributed patches were either ignored or put up on the MySQL++ web site for users to try, without any official blessing.

The biggest barrier to using MySQL++ during this period is that the popular C++ compilers of 2001 weren't all that compatible with the C++ Standard. As a result, MySQL++ used many nonstandard constructs, to allow for compatibility with older compilers. Each new compiler released in the following years increased compliance, either warning about or rejecting code using pre-Standard constructs. In particular, GCC was emerging from the mess following the EGCS fork during this time. The fork was healed officially in 1999, but there's always a delay of a few years between the release of a new GCC and widespread adoption. The post-EGCS versions of GCC were only beginning to become popular by 2001, when development on MySQL++ halted. As a result, it became increasingly difficult to get MySQL++ to build cleanly as newer compilers came out. Since MySQL++ uses templates heavily, this affected end user programs as well: MySQL++ code got included directly in your program, so any warnings or errors it caused became your program's problem.

As a result, most of the patches contributed to the MySQL++ project during this period were to fix up standards compliance issues. Because no one was bothering to officially test and bless these patches, you ended up with the worst aspects of a bazaar development model: complete freedom of development, but no guiding hand to select from the good stuff and reject the rest. Many of the patches were mutually incompatible. Some would build upon other patches, so you had to apply them in the proper sequence. Others did useful things, but didn't give a fully functional copy of MySQL++. Figuring out which patch(es) to use was an increasingly frustrating exercise as the years wore on, and newer GCCs became popular.

In early August of 2004, Warren Young got fed up with this situation and took over. He released 1.7.10 later that month, which did little more than make the code build with GCC 3.3 without warnings. Since then, with a little help from his friends on the Net, MySQL++ has lost a lot of bugs, gained a lot of features, gained a few more bugs, lost them again... MySQL++ is alive and healthy now.

[1] The MySQL C API is also known as Connector/C.

## 1.2. If You Have Questions...

If you want to email someone to ask questions about this library, we greatly prefer that you send mail to the MySQL++ mailing list. The mailing list is archived, so if you have questions, do a search to see if the question has been asked before.

You may find people's individual email addresses in various files within the MySQL++ distribution. Please do not send mail to them unless you are sending something that is inherently personal. Not all of the principal developers of MySQL++ are still active in its development; those who have dropped out have no wish to be bugged about MySQL++. Those of us still active in MySQL++ development monitor the mailing list, so you aren't getting any extra "coverage" by sending messages to additional email addresses.

# 2. Overview

MySQL++ has a lot of complexity and power to cope with the variety of ways people use databases, but at bottom it doesn't work all that differently than other database access APIs. The usage pattern looks like this:

1. Open the connection
2. Form and execute the query
3. If successful, iterate through the result set
4. Else, deal with errors

Each of these steps corresponds to a MySQL++ class or class hierarchy. An overview of each follows.

## 2.1. The Connection Object

A Connection object manages the connection to the MySQL server. You need at least one of these objects to do anything. Because the other MySQL++ objects your program will use often depend (at least indirectly) on the `Connection` instance, the `Connection` object needs to live at least as long as all other MySQL++ objects in your program.

MySQL supports many different types of data connection between the client and the server: TCP/IP, Unix domain sockets, and Windows named pipes. The generic `Connection` class supports all of these, figuring out which one you mean based on the parameters you pass to `Connection::connect()`. But if you know in advance that your program only needs one particular connection type, there are subclasses with simpler interfaces. For example, there's TCPConnection if you know your program will always use a networked database server.

## 2.2. The Query Object

Most often, you create SQL queries using a Query object created by the `Connection` object.

`Query` acts as a standard C++ output stream, so you can write data to it like you would to `std::cout` or `std::ostringstream`. This is the most C++ish way MySQL++ provides for building up a query string. The library includes stream manipulators that are type-aware so it's easy to build up syntactically-correct SQL.

`Query` also has a feature called Template Queries which work something like C's `printf()` function: you set up a fixed query string with tags inside that indicate where to insert the variable parts. If you have multiple queries that are structurally similar, you simply set up one template query, and use that in the various locations of your program.

A third method for building queries is to use `Query` with SSQLS. This feature lets you create C++ structures that mirror your database schemas. These in turn give `Query` the information it needs to build many common SQL queries for you. It can **INSERT**, **REPLACE** and **UPDATE** rows in a table given the data in SSQLS form. It can also generate **SELECT * FROM SomeTable** queries and store the results as an STL collection of SSQLSes.

## 2.3. Result Sets

The field data in a result set are stored in a special `std::string`-like class called String. This class has conversion operators that let you automatically convert these objects to any of the basic C data types. Additionally, MySQL++ defines classes like DateTime, which you can initialize from a MySQL **DATETIME** string. These automatic conversions are protected against bad conversions, and can either set a warning flag or throw an exception, depending on how you set the library up.

As for the result sets as a whole, MySQL++ has a number of different ways of representing them:

## Queries That Do Not Return Data

Not all SQL queries return data. An example is **CREATE TABLE**. For these types of queries, there is a special result type (SimpleResult) that simply reports the state resulting from the query: whether the query was successful, how many rows it impacted (if any), etc.

## Queries That Return Data: MySQL++ Data Structures

The most direct way to retrieve a result set is to use `Query::store()`. This returns a StoreQueryResult object, which derives from `std::vector<mysqlpp::Row>`, making it a random-access container of Rows. In turn, each `Row` object is like a `std::vector` of `String` objects, one for each field in the result set. Therefore, you can treat `StoreQueryResult` as a two-dimensional array: you can get the 5th field on the 2nd row by simply saying `result[1][4]`. You can also access row elements by field name, like this: `result[2]["price"]`.

A less direct way of working with query results is to use `Query::use()`, which returns a UseQueryResult object. This class acts like an STL input iterator rather than a `std::vector`: you walk through your result set processing one row at a time, always going forward. You can't seek around in the result set, and you can't know how many results are in the set until you find the end. In payment for that inconvenience, you get better memory efficiency, because the entire result set doesn't need to be stored in RAM. This is very useful when you need large result sets.

## Queries That Return Data: Specialized SQL Structures

Accessing results through MySQL++'s data structures is a pretty low level of abstraction. It's better than using the MySQL C API, but not by much. You can elevate things a little closer to the level of the problem space by using the SSQLS feature. This lets you define C++ structures that match the table structures in your database schema. In addition, it's easy to use SSQLSes with regular STL containers (and thus, algorithms) so you don't have to deal with the quirks of MySQL++'s data structures.

The advantage of this method is that your program will require very little embedded SQL code. You can simply execute a query, and receive your results as C++ data structures, which can be accessed just as you would any other structure. The results can be accessed through the Row object, or you can ask the library to dump the results into an STL container — sequential or set-associative, it doesn't matter — for you. Consider this:

```
vector<stock> v;
query << "SELECT * FROM stock";
query.storein(v);
for (vector<stock>::iterator it = v.begin(); it != v.end(); ++it) {
  cout << "Price: " << it->price << endl;
}
```

Isn't that slick?

If you don't want to create SSQLSes to match your table structures, as of MySQL++ v3 you can now use `Row` here instead:

```
vector<mysqlpp::Row> v;
query << "SELECT * FROM stock";
query.storein(v);
for (vector<mysqlpp::Row>::iterator it = v.begin(); it != v.end(); ++it) {
  cout << "Price: " << it->at("price") << endl;
}
```

It lacks a certain syntactic elegance, but it has its uses.

## 2.4. Exceptions

By default, the library throws exceptions whenever it encounters an error. You can ask the library to set an error flag instead, if you like, but the exceptions carry more information. Not only do they include a string member telling you why the exception was thrown, there are several exception types, so you can distinguish between different error types within a single try block.

# 3. Tutorial

The previous chapter introduced the major top-level mechanisms in MySQL++. Now we'll dig down a little deeper and get into real examples. We start off with the basics that every MySQL++ program will have to deal with, then work up to more complex topics that are still widely interesting. You can stop reading the manual after this chapter and still get a lot out of MySQL++, ignoring the more advanced parts we present in later chapters.

## 3.1. Running the Examples

All of the examples are complete running programs. If you built the library from source, the examples should have been built as well. If you use RPMs instead, the example programs' source code and a simplified `Makefile` are in the `mysql++-devel` package. They are typically installed in `/usr/share/doc/mysql++-devel-*/examples`, but it can vary on different Linuxes.

Before you get started, please read through any of the `README*.txt` files included with the MySQL++ distribution that are relevant to your platform. We won't repeat all of that here.

Most of the examples require a test database, created by `resetdb`. You can run it like so:

```
resetdb [-s server_addr] [-u user] [-p password]
```

Actually, there's a problem with that. It assumes that the MySQL++ library is already installed in a directory that the operating system's dynamic linker can find. (MySQL++ is almost never built statically.) Unless you're installing from RPMs, you've had to build the library from source, and you should run at least a few of the examples before installing the library to be sure it's working correctly. Since your operating system's dynamic linkage system can't find the MySQL++ libraries without help until they're installed, we've created a few helper scripts to help run the examples.

MySQL++ comes with the `exrun` shell script for Unixy systems, and the `exrun.bat` batch file for Windows. You pass the example program and its arguments to the `exrun` helper, which sets up the library search path so that it will use the as-yet uninstalled version of the MySQL++ library in preference to any other on your system:

```
./exrun resetdb [-s server_addr] [-u user] [-p password]
```

That's the typical form for a Unixy system. You leave off the **./** bit on Windows. You can leave it off on a Unixy system, too, if you have `.` in your `PATH`. (Not a recommendation, just an observation.)

All of the program arguments are optional.

If you don't give `-s`, the underlying MySQL C API (a.k.a. Connector/C) assumes the server is on the local machine. It chooses one of several different IPC options based on the platform configuration. There are many different forms you can give as `server_addr` with `-s` to override this default behavior:

- *localhost* — this is the default; it doesn't buy you anything
- On Windows, a simple period tells the underlying MySQL C API to use named pipes, if it's available.
- *172.20.0.252:12345* — this would connect to IP address `172.20.0.252` on TCP port `12345`.
- *my.server.name:svc_name* — this would first look up TCP service name `svc_name` in your system's network services database (`/etc/services` on Unixy systems, and something like `c:\windows\system32\drivers\etc\services` on modern Windows variants). If it finds an entry for the service, it then tries to connect to that port on the domain name given.

For the TCP forms, you can mix names and numbers for the host and port/service parts in any combination. If the server name doesn't contain a colon, it uses the default port, 3306.

If you don't give `-u`, it assumes your user name on the database server is the same as your login name on the local machine.

If you don't give `-p`, it will assume the MySQL user doesn't have a password. (One hopes this isn't the case...)

When running `resetdb`, the user name needs to be for an account with permission to create the test database. Once the database is created, you can use any account when running the other examples that has DELETE, INSERT, SELECT and UPDATE permissions for the test database. The MySQL root user can do all this, of course, but you might want to set up a separate user, having only the permissions necessary to work with the test database:

```
CREATE USER mysqlpp_test@'%' IDENTIFIED BY 'nunyabinness';
GRANT ALL PRIVILEGES ON mysql_cpp_data.* TO mysqlpp_test@'%';
```

You could then create the sample database with the following command:

```
./exrun resetdb -u mysqlpp_test -p nunyabinness
```

(Again, leave off the **./** bit on Windows.)

You may have to re-run `resetdb` after running some of the other examples, as they change the database.

See `README-examples.txt` for more details on running the examples.

# 3.2. A Simple Example

The following example demonstrates how to open a connection, execute a simple query, and display the results. This is `examples/simple1.cpp`:

```
#include "cmdline.h"
#include "printdata.h"

#include <mysql++.h>

#include <iostream>
#include <iomanip>

using namespace std;

int
main(int argc, char *argv[])
{
    // Get database access parameters from command line
    mysqlpp::examples::CommandLine cmdline(argc, argv);
    if (!cmdline) {
        return 1;
    }

    // Connect to the sample database.
    mysqlpp::Connection conn(false);
    if (conn.connect(mysqlpp::examples::db_name, cmdline.server(),
            cmdline.user(), cmdline.pass())) {
        // Retrieve a subset of the sample stock table set up by resetdb
        // and display it.
        mysqlpp::Query query = conn.query("select item from stock");
        if (mysqlpp::StoreQueryResult res = query.store()) {
            cout << "We have:" << endl;
            mysqlpp::StoreQueryResult::const_iterator it;
            for (it = res.begin(); it != res.end(); ++it) {
                mysqlpp::Row row = *it;
```

```
                cout << '\t' << row[0] << endl;
            }
        }
        else {
            cerr << "Failed to get item list: " << query.error() << endl;
            return 1;
        }

        return 0;
    }
    else {
        cerr << "DB connection failed: " << conn.error() << endl;
        return 1;
    }
}
```

This example simply gets the entire "item" column from the example table, and prints those values out.

Notice that MySQL++'s StoreQueryResult derives from `std::vector`, and Row provides an interface that makes it a `vector` work-alike. This means you can access elements with subscript notation, walk through them with iterators, run STL algorithms on them, etc.

`Row` provides a little more in this area than a plain old `vector`: you can also access fields by name using subscript notation.

The only thing that isn't explicit in the code above is that we delegate command line argument parsing to `parse_command_line()` in the `examples` module. This function exists to give the examples a consistent interface, not to hide important details. You can treat it like a black box: it takes `argc` and `argv` as inputs and sends back database connection parameters.

# 3.3. A More Complicated Example

The `simple1` example above was pretty trivial. Let's get a little deeper. Here is `examples/simple2.cpp`:

```
#include "cmdline.h"
#include "printdata.h"

#include <mysql++.h>

#include <iostream>
#include <iomanip>

using namespace std;

int
main(int argc, char *argv[])
{
    // Get database access parameters from command line
    mysqlpp::examples::CommandLine cmdline(argc, argv);
    if (!cmdline) {
        return 1;
    }

    // Connect to the sample database.
    mysqlpp::Connection conn(false);
    if (conn.connect(mysqlpp::examples::db_name, cmdline.server(),
            cmdline.user(), cmdline.pass())) {
        // Retrieve the sample stock table set up by resetdb
        mysqlpp::Query query = conn.query("select * from stock");
        mysqlpp::StoreQueryResult res = query.store();
```

```
        // Display results
        if (res) {
            // Display header
            cout.setf(ios::left);
            cout << setw(31) << "Item" <<
                    setw(10) << "Num" <<
                    setw(10) << "Weight" <<
                    setw(10) << "Price" <<
                    "Date" << endl << endl;

            // Get each row in result set, and print its contents
            for (size_t i = 0; i < res.num_rows(); ++i) {
                cout << setw(30) << res[i]["item"] << ' ' <<
                        setw(9) << res[i]["num"] << ' ' <<
                        setw(9) << res[i]["weight"] << ' ' <<
                        setw(9) << res[i]["price"] << ' ' <<
                        setw(9) << res[i]["sdate"] <<
                        endl;
            }
        }
        else {
            cerr << "Failed to get stock table: " << query.error() << endl;
            return 1;
        }

        return 0;
    }
    else {
        cerr << "DB connection failed: " << conn.error() << endl;
        return 1;
    }
}
```

The main point of this example is that we're accessing fields in the row objects by name, instead of index. This is slower, but obviously clearer. We're also printing out the entire table, not just one column.

# 3.4. Exceptions

By default, MySQL++ uses exceptions to signal errors. We've been suppressing this in all the examples so far by passing false to `Connection`'s constructor. This kept these early examples simple at the cost of some flexibility and power in error handling. In a real program, we recommend that you leave exceptions enabled. You do this by either using the default `Connection` constructor, or by using the create-and-connect constructor.

All of MySQL++'s custom exceptions derive from a common base class, Exception. That in turn derives from Standard C++'s `std::exception` class. Since the library can indirectly cause exceptions to come from the Standard C++ Library, it's possible to catch all exceptions from MySQL++ by just catching `std::exception`. However, it's better to have individual catch blocks for each of the concrete exception types that you expect, and add a handler for either `Exception` or `std::exception` to act as a "catch-all" for unexpected exceptions.

When exceptions are suppressed, MySQL++ signals errors by returning either an error code or an object that tests as false, or by setting an error flag on the object. Classes that allow you to suppress exceptions derive from the OptionalExceptions interface. When an `OptionalExceptions` derivative creates another object that also derives from this interface, it passes on its exception flag. Since everything flows from the Connection object, disabling exceptions on it at the start of the program disables all optional exceptions. This is why passing false for the `Connection` constructor's "throw exceptions" parameter suppresses all optional exceptions in the `simple[1-3]` examples. It keeps them, well, simple.

This exception suppression mechanism is quite granular. It's possible to leave exceptions enabled most of the time, but suppress them in sections of the code where they aren't helpful. To do this, put the section of code that you want to not throw exceptions inside a block, and create a NoExceptions object at the top of that block. When created, it saves the exception flag of the `OptionalExceptions` derivative you pass to it, and then disables exceptions on it. When the `NoExceptions` object goes out of scope at the end of the block, it restores the exceptions flag to its previous state:

```
mysqlpp::Connection con; // default ctor, so exceptions enabled

{
  mysqlpp::NoExceptions ne(con);
  if (!con.select_db("a_db_that_might_not_exist_yet")) {
    // Our DB doesn't exist yet, so create and select it here; no need
    // to push handling of this case way off in an exception handler.
  }
}
```

When one `OptionalExceptions` derivative passes its exceptions flag to another such object, it is only passing a copy; the two objects' flags operate independently. There's no way to globally enable or disable this flag on existing objects in a single call. If you're using the `NoExceptions` feature and you're still seeing optional exceptions thrown, you disabled exceptions on the wrong object. The exception thrower could be unrelated to the object you disabled exceptions on, it could be its parent, or it could be a child created before you disabled optional exceptions.

MySQL++ throws some exceptions unconditionally:

- MySQL++ checks array indices, always. For instance, if your code said "`row[21]`" on a row containing only 5 fields, you'd get a `BadIndex` exception. If you say "`row["fred"]`" on a row without a "fred" field, you get a `BadFieldName` exception. In the past, MySQL++ delegated some of its index checking to the STL containers underpinning it, so you could get `std::range_error` instead. As of MySQL++ v3.0.7, this should no longer happen, but there may be instances where it still does.

- String will always throw BadConversion when you ask it to do an improper type conversion. For example, you'll get an exception if you try to convert "1.25" to int, but not when you convert "1.00" to int. In the latter case, MySQL++ knows that it can safely throw away the fractional part.

- If you use template queries and don't pass enough parameters when instantiating the template, `Query` will throw a BadParamCount exception.

- If you use a C++ data type in a query that MySQL++ doesn't know to convert to SQL, MySQL++ will throw a TypeLookupFailed exception. It typically happens with Section 5, "Specialized SQL Structures", especially when using data types other than the ones defined in `lib/sql_types.h`.

It's educational to modify the examples to force exceptions. For instance, misspell a field name, use an out-of-range index, or change a type to force a `String` conversion error.

# 3.5. Quoting and Escaping

SQL syntax often requires certain data to be quoted. Consider this query:

```
SELECT * FROM stock WHERE item = 'Hotdog Buns'
```

Because the string "Hotdog Buns" contains a space, it must be quoted. With MySQL++, you don't have to add these quote marks manually:

```
string s = "Hotdog Buns";
query << "SELECT * FROM stock WHERE item = " << quote_only << s;
```

That code produces the same query string as in the previous example. We used the MySQL++ quote_only manipulator, which causes single quotes to be added around the next item inserted into the stream. This works for any type of data that can be converted to MySQL++'s SQLTypeAdapter type, plus the Set template. SSQLS also uses these manipulators internally.

Quoting is pretty simple, but SQL syntax also often requires that certain characters be "escaped". Imagine if the string in the previous example was "Frank's Brand Hotdog Buns" instead. The resulting query would be:

```
SELECT * FROM stock WHERE item = 'Frank's Brand Hotdog Buns'
```

That's not valid SQL syntax. The correct syntax is:

```
SELECT * FROM stock WHERE item = 'Frank''s Brand Hotdog Buns'
```

As you might expect, MySQL++ provides that feature, too, through its escape manipulator. But here, we want both quoting and escaping. That brings us to the most widely useful manipulator:

```
string s = "Frank's Brand Hotdog Buns";
query << "SELECT * FROM stock WHERE item = " << quote << s;
```

The quote manipulator both quotes strings and escapes any characters that are special in SQL.

MySQL++ provides other manipulators as well. See the manip.h page in the reference manual.

It's important to realize that MySQL++'s quoting and escaping mechanism is type-aware. Manipulators have no effect unless you insert the manipulator into a `Query` or SQLQueryParms stream. [2] Also, values are only quoted and/or escaped if they are of a data type that may need it. For example, Date must be quoted but never needs to be escaped, and integer types need neither quoting nor escaping. Manipulators are suggestions to the library, not commands: MySQL++ will ignore these suggestions if it knows it won't result in syntactically-incorrect SQL.

It's also important to realize that quoting and escaping in `Query` streams and template queries is never implicit.[3] You must use manipulators and template query flags as necessary to tell MySQL++ where quoting and escaping is necessary. It would be nice if MySQL++ could do quoting and escaping implicitly based on data type, but this isn't possible in all cases.[4] Since MySQL++ can't reliably guess when quoting and escaping is appropriate, and the programmer doesn't need to[5], MySQL++ makes you tell it.

# 3.6. C++ vs. SQL Data Types

The C++ and SQL data type systems have several differences that can cause problems when using MySQL++, or any other SQL based system, for that matter.

[2] `SQLQueryParms` is used as a stream only as an implementation detail within the library. End user code simply sees it as a `std::vector` derivative.

[3] By contrast, the `Query` methods that take an SSQLS *do* add quotes and escape strings implicitly. It can do this because SSQLS knows all the SQL code and data types, so it never has to guess whether quoting or escaping is appropriate.

[4] Unless you're smarter than I am, you don't immediately see why explicit manipulators are necessary. We can tell when quoting and escaping is *not* appropriate based on type, so doesn't that mean we know when it *is* appropriate? Alas, no. For most data types, it is possible to know, or at least make an awfully good guess, but it's a complete toss-up for C strings, const char*. A C string could be either a literal string of SQL code, or it can be a value used in a query. Since there's no easy way to know and it would damage the library's usability to mandate that C strings only be used for one purpose or the other, the library requires you to be explicit.

[5] One hopes the programmer *knows*.

Most of the data types you can store in a SQL database are either numbers or text strings. If you're only looking at the data going between the database server and your application, there aren't even numbers: SQL is a textual language, so numbers and everything else gets transferred between the client and the database server in text string form.[6] Consequently, MySQL++ has a lot of special support for text strings, and can translate to several C++ numeric data types transparently.

Some people worry that this translation via an intermediate string form will cause data loss. Obviously the text string data types are immune from problems in this regard. We're also confident that MySQL++ translates BLOB and integer data types losslessly.

The biggest worry is with floating-point numbers. (The FLOAT and DOUBLE SQL data types.) We did have a problem with this in older versions of MySQL++, but we believe we fixed it completely in v3.0.2. No one has since proven data loss via this path. There is still a known problem [7] with the SQL DECIMAL type, which is somewhat related to the floating-point issue, but it's apparently rarely encountered, which is why it hasn't been fixed yet.

The best way to avoid problems with data translation is to always use the special MySQL++ data types defined in `lib/sql_types.h` corresponding to your SQL schema. These typedefs begin with sql_ and end with a lowercase version of the standard SQL type name, with spaces replaced by underscores. There are variants ending in _null that wrap these base types so they're compatible with SQL null. For instance, the SQL type TINYINT UNSIGNED NOT NULL is represented in MySQL++ by `mysqlpp::sql_tinyint_unsigned`. If you drop the NOT NULL part, the corresponding C++ type is `mysqlpp::sql_tinyint_unsigned_null`.

MySQL++ doesn't force you to use these typedefs. It tries to be flexible with regard to data conversions, so you could probably use int anywhere you use `mysqlpp::sql_tinyint_unsigned`, for example. That said, the MySQL++ typedefs give several advantages:

- Space efficiency: the MySQL++ types are no larger than necessary to hold the MySQL data.
- Portability: if your program has to run on multiple different system types (even just 32- and 64-bit versions of the same operating system and processor type) using the MySQL++ typedefs insulates your code from platform changes.
- Clarity: using C++ types named similarly to the SQL types reduces the risk of confusion when working with code in both languages at the same time.
- Compatibility: using the MySQL++ types ensures that data conversions between SQL and C++ forms are compatible. Naïve use of plain old C++ types can result in data truncation, TypeLookupFailed exceptions, and worse.

  Type compatibility is important not just at the time you write your program, it also helps forward compatibility: we occasionally change the definitions of the MySQL++ typedefs to reduce the differences between the C++ and SQL type systems. We'll be fixing the DECIMAL issue brought up above this way, for instance; if your program uses `sql_decimal` instead of the current underlying type, double, your program will pick up this improvement automatically with just a recompile.

Most of these typedefs use standard C++ data types, but a few are aliases for a MySQL++ specific type. For instance, the SQL type `DATETIME` is mirrored in MySQL++ by `mysqlpp::DateTime`. For consistency, `sql_types.h` includes a typedef alias for `DateTime` called `mysqlpp::sql_datetime`.

[6] Yes, we're aware that there is a feature in MySQL that lets you transfer row data in a binary form, but we don't support this yet. We may, someday, probably as an extension to SSQLS. The only real reason to do so is to shave off some of the data translation overhead, which is typically neglibible in practice, swamped by the far greater disk and network I/O overheads inherent in use of a client-server database system like MySQL.

[7] SQL's DECIMAL data type is a configurable-precision fixed-point number format. MySQL++ currently translates these to double, a floating-point data format, the closest thing available in the C++ type system. Since the main reason to use DECIMAL is to get away from the weird roundoff behavior of floating-point numbers, this could be viewed as a serious problem. The thing is, though, in all the years MySQL++ has been around, I don't remember anyone actually complaining about it. Apparently there's either no one using DECIMAL with MySQL++, or they're ignoring any roundoff errors they get as a result. Until this wheel squeaks, it's not likely to be greased. To fix this, we'll have to create a new custom data type to hold such column values, which will be a lot of work for apparently little return.

MySQL++ doesn't have typedefs for the most exotic data types, like those for the geospatial types. Patches to correct this will be thoughtfully considered.

# 3.7. Handling SQL Nulls

Both C++ and SQL have things in them called NULL, but they differ in several ways. Consequently, MySQL++ has to provide special support for this, rather than just wrap native C++ facilities as it can with most data type issues.

## SQL NULL is a type modifier

The primary distinction is one of type. In SQL, "NULL" is a type modifier, which affects whether you can legally store a null value in that column. There's simply nothing like it in C++.

To emulate SQL NULL, MySQL++ provides the Null template to allow the creation of distinct "nullable" versions of existing C++ types. So for example, if you have a TINYINT UNSIGNED column that can have nulls, the proper declaration for MySQL++ would be:

```
mysqlpp::Null<mysqlpp::sql_tinyint_unsigned> myfield;
```

As of MySQL++ 3.1, we also provide shorter aliases for such types:

```
mysqlpp::sql_tinyint_unsigned_null myfield;
```

These types are declared in `lib/sql_types.h`. You might want to scan through that to see what all is available.

Template instantiations are first-class types in the C++ language, so there's no possible confusion between this feature of MySQL++ and C++'s native NULL concept.

## SQL NULL is a unique value

There's a secondary distinction between SQL null and anything available in the standard C++ type system: SQL null is a distinct value, equal to nothing else. We can't use C++'s NULL for this because it is ambiguous, being equal to 0 in integer context. MySQL++ provides the global `null` object, which you can assign to a `Null` template instance to make it equal to SQL null:

```
myfield = mysqlpp::null;
```

If you insert a MySQL++ field holding a SQL null into a C++ IOstream, you get "(NULL)", something fairly unlikely to be in a normal output string, thus reasonably preserving the uniqueness of the SQL null value.

MySQL++ also tries to enforce the uniqueness of the SQL null value at compile time in assignments and data conversions. If you try to store a SQL null in a field type that isn't wrapped by `Null` or try to assign a `Null`-wrapped field value to a variable of the inner non-wrapped type, the compiler will emit some ugly error message, yelling about CannotConvertNullToAnyOtherDataType. (The exact message is compiler-dependent.)

If you don't like these behaviors, you can change them by passing a different value for the second parameter to template `Null`. By default, this parameter is NullIsNull, meaning that we should enforce the uniqueness of SQL null. To relax the distinctions, you can instantiate the `Null` template with a different behavior type: NullIsZero or NullIsBlank. Consider this code:

```
mysqlpp::Null<unsigned char, mysqlpp::NullIsZero> myfield(mysqlpp::null);
cout << myfield << endl;
cout << int(myfield) << endl;
```

This will print “0” twice. If you had used the default for the second `Null` template parameter, the first output statement would have printed “(NULL)”, and the second wouldn’t even compile.

# 3.8. MySQL++’s Special String Types

MySQL++ has two classes that work like `std::string` to some degree: String and SQLTypeAdapter. These classes exist to provide functionality that `std::string` doesn’t provide, but they are neither derivatives of nor complete supersets of `std::string`. As a result, end-user code generally doesn’t deal with these classes directly, because `std::string` is a better general-purpose string type. In fact, MySQL++ itself uses `std::string` most of the time, too. But, the places these specialized stringish types do get used are so important to the way MySQL++ works that it’s well worth taking the time to understand them.

## SQLTypeAdapter

The simpler of the two is `SQLTypeAdapter`, or `STA` for short.[8]

As its name suggests, its only purpose is to adapt other data types to be used with SQL. It has a whole bunch of conversion constructors, one for all data types we expect to be used with MySQL++ for values in queries. SQL queries are strings, so constructors that take stringish types just make a copy of that string, and all the others “stringize” the value in the format needed by SQL.[9] The conversion constructors preserve type information, so this stringization process doesn’t throw away any essential information.

`STA` is used anywhere MySQL++ needs to be able to accept any of several data types for use in a SQL query. Major users are `Query`’s template query mechanism and the `Query` stream quoting and escaping mechanism. You care about `STA` because any time you pass a data value to MySQL++ to be used in building a SQL query, it goes through `STA`. `STA` is one of the key pieces in MySQL++ that makes it easy to generate syntactically-correct SQL queries.

## String

If MySQL++ can be said to have its own generic string type, it’s `String`, but it’s not really functional enough for general use. It’s possible that in future versions of MySQL++ we’ll expand its interface to include everything `std::string` does, so that’s why it’s called that.[10]

The key thing `String` provides over `std::string` is conversion of strings in SQL value formats to their plain old C++ data types. For example, if you initialize it with the string “2007-11-19”, you can assign the `String` to a Date, not because `Date` knows how to initialize itself from `String`, but the reverse: `String` has a bunch of implicit conversion operators defined for it, so you can use it in any type context that makes sense in your application.

Because `Row::operator[]` returns `String`, you can say things like this:

```
int x = row["x"];
```

In a very real sense, `String` is the inverse of `STA`: `String` converts SQL value strings to C++ data types, and `STA` converts C++ data types to SQL value strings.[11]

[8] In version 2 of MySQL++ and earlier, `SQLTypeAdapter` was called `SQLString`, but it was confusing because its name and the fact that it derived from `std::string` suggested that it was a general-purpose string type. MySQL++ even used it this way in a few places internally. In v3, we made it a simple base class and renamed it to reflect its proper limited function.

[9] `SQLTypeAdapter` doesn’t do quoting and escaping itself. That happens elsewhere, right at the point that the `STA` gets used to build a query.

[10] If you used MySQL++ before v3, `String` used to be called `ColData`. It was renamed because starting in v2.3, we began using it for holding more than just column data. I considered renaming it `SQLString` instead, but that would have confused old MySQL++ users to no end. Instead, I followed the example of `Set`, MySQL++’s specialized `std::set` variant.

[11] During the development of MySQL++ v3.0, I tried merging `SQLTypeAdapter` and `String` into a single class to take advantage of this. The resulting class gave the C++ compiler the freedom to tie itself up in knots, because it was then allowed to convert almost any data type to almost any other. You’d get a tangle of ambiguous data type conversion errors from the most innocent code.

`String` has two main uses.

By far the most common use is as the field value type of `Row`, as exemplified above. It's not just the return type of `Row::operator[]`, though: it's actually the value type used within `Row`'s internal array. As a result, any time MySQL++ pulls data from the database, it goes through `String` when converting it from the string form used in SQL result sets to the C++ data type you actually want the data in. It's the core of the structure population mechanism in the SSQLS feature, for example.

Because `String` is the last pristine form of data in a result set before it gets out of MySQL++'s internals where end-user code can see it, MySQL++'s sql_blob and related typedefs are aliases for `String`. Using anything else would require copies; while the whole "networked database server" thing means most of MySQL++ can be quite inefficient and still not affect benchmark results meaningfully, BLOBs tend to be big, so making unnecessary copies can really make a difference. Which brings us to...

## Reference Counting

To avoid unnecessary buffer copies, both `STA` and `String` are implemented in terms of a reference-counted copy-on-write buffer scheme. Both classes share the same underlying mechanism, and so are interoperable. This means that if you construct one of these objects from another, it doesn't actually copy the string data, it only copies a pointer to the data buffer, and increments its reference count. If the object has new data assigned to it or it's otherwise modified, it decrements its reference count and creates its own copy of the buffer. This has a lot of practical import, such as the fact that even though `Row::operator[]` returns `String`s by value, it's still efficient.

# 3.9. Dealing with Binary Data

Historically, there was no way to hold arbitrary-sized blocks of raw binary data in an SQL database. There was resistance to adding such a feature to SQL for a long time because it's better, where possible, to decompose blocks of raw binary data into a series of numbers and text strings that *can* be stored in the database. This lets you query, address and manipulate elements of the data block individually.

A classic SQL newbie mistake is trying to treat the database server as a file system. Some embedded platforms use a database engine as a file system, but MySQL doesn't typically live in that world. When your platform already has a perfectly good file system, you should use it for big, nondecomposable blocks of binary data in most cases.

A common example people use when discussing this is images in database-backed web applications. If you store the image in the database, you have to write code to retrieve the image from the database and send it to the client; there's more overhead, and less efficient use of the system's I/O caching system. If you store the image in the filesystem, all you have to do is point the web server to the directory where the images live, and put a URL for that image in your generated HTML. Because you're giving the web server a direct path to a file on disk, operation is far more efficient. Web servers are very good at slurping whole files off of disk and sending them out to the network, and operating systems are very good at caching file accesses. Plus, you avoid the overhead of pushing the data through the high-level language your web app is written in, which is typically an interpreted language, not C++. Some people still hold out on this, claiming that database engines have superior security features, but I call bunk on that, too. Operating systems and web servers are capable of building access control systems every bit as granular and secure as a database system.

Occasionally you really do need to store a nondecomposable block of binary data in the database. For such cases, modern SQL database servers support BLOB data types, for Binary Large OBject. This is often just called binary data, though of course all data in a modern computer is binary at some level.

The tricky part about dealing with binary data in MySQL++ is to ensure that you don't ever treat the data as a C string, which is really easy to do accidentally. C strings treat zero bytes as special end-of-string characters, but they're not special at all in binary data. We've made a lot of improvements to the way MySQL++ handles string data to avoid this problem, but it's still possible to bypass these features, wrecking your BLOBs. These examples demonstrate correct techniques.

## Loading a binary file into a BLOB column

Above, I opined that it's usually incorrect to store image data in a database, particularly with web apps, of which CGI is a primitive form. Still, it makes a nice, simple example.

Instead of a single example program, we have here a matched pair. The first example takes the name of a JPEG file on the command line along with all the other common example program parameters, loads that file into memory, and stores it in a BLOB column in the database.

This example also demonstrates how to retrieve the value assigned to an auto-increment column in the previous insertion. This example uses that feature in the typical way, to create unique IDs for rows as they're inserted.

Here is `examples/load_jpeg.cpp`:

```
#include "cmdline.h"
#include "images.h"
#include "printdata.h"

#include <fstream>

using namespace std;
using namespace mysqlpp;


// This is just an implementation detail for the example.  Skip down to
// main() for the concept this example is trying to demonstrate.  You
// can simply assume that, given a BLOB containing a valid JPEG, it
// returns true.
static bool
is_jpeg(const mysqlpp::sql_blob& img, const char** whynot)
{
    // See http://stackoverflow.com/questions/2253404/ for
    // justification for the various tests.
    const unsigned char* idp =
            reinterpret_cast<const unsigned char*>(img.data());
    if (img.size() < 125) {
        *whynot = "a valid JPEG must be at least 125 bytes";
    }
    else if ((idp[0] != 0xFF) || (idp[1] != 0xD8)) {
        *whynot = "file does not begin with JPEG sigil bytes";
    }
    else if ((memcmp(idp + 6, "JFIF", 4) != 0) &&
             (memcmp(idp + 6, "Exif", 4) != 0)) {
        *whynot = "file does not contain JPEG type word";
    }
    else {
        *whynot = 0;
        return true;
    }

    return false;
}


// Skip to main() before studying this.  This is a little too
// low-level to bother with on your first pass thru the code.
static bool
load_jpeg_file(const mysqlpp::examples::CommandLine& cmdline,
        images& img, string& img_name)
{
    if (cmdline.extra_args().size() == 0) {
```

```
        // Nothing for us to do here.  Caller will insert NULL BLOB.
        return true;
    }

    // Got a file's name on the command line, so open it.
    img_name = cmdline.extra_args()[0];
    ifstream img_file(img_name.c_str(), ios::binary);
    if (img_file) {
        // Slurp file contents into RAM with minimum copying.  (Idiom
        // explained here: http://stackoverflow.com/questions/116038/)
        //
        // By loading the file into a C++ string (stringstream::str())
        // and assigning that directly to a mysqlpp::sql_blob, we avoid
        // truncating the binary data at the first null character.
        img.data.data = static_cast<const stringstream*>(
                &(stringstream() << img_file.rdbuf()))->str();

        // Check JPEG data for sanity.
        const char* error;
        if (is_jpeg(img.data.data, &error)) {
            return true;
        }
        else {
            cerr << '"' << img_name << "\" isn't a JPEG: " <<
                    error << '!' << endl;
        }
    }

    cmdline.print_usage("[jpeg_file]");
    return false;
}


int
main(int argc, char *argv[])
{
    // Get database access parameters from command line
    mysqlpp::examples::CommandLine cmdline(argc, argv);
    if (!cmdline) {
        return 1;
    }

    try {
        // Establish the connection to the database server.
        mysqlpp::Connection con(mysqlpp::examples::db_name,
                cmdline.server(), cmdline.user(), cmdline.pass());

        // Load the file named on the command line
        images img(mysqlpp::null, mysqlpp::null);
        string img_name("NULL");
        if (load_jpeg_file(cmdline, img, img_name)) {
            // Insert image data or SQL NULL into the images.data BLOB
            // column.  The key here is that we're holding the raw
            // binary data in a mysqlpp::sql_blob, which avoids data
            // conversion problems that can lead to treating BLOB data
            // as C strings, thus causing null-truncation.  The fact
            // that we're using SSQLS here is a side issue, simply
            // demonstrating that mysqlpp::Null<mysqlpp::sql_blob> is
            // now legal in SSQLS, as of MySQL++ 3.0.7.
            Query query = con.query();
            query.insert(img);
            SimpleResult res = query.execute();
```

```
            // Report successful insertion
            cout << "Inserted \"" << img_name <<
                    "\" into images table, " << img.data.data.size() <<
                    " bytes, ID " << res.insert_id() << endl;
        }
    }
    catch (const BadQuery& er) {
        // Handle any query errors
        cerr << "Query error: " << er.what() << endl;
        return -1;
    }
    catch (const BadConversion& er) {
        // Handle bad conversions
        cerr << "Conversion error: " << er.what() << endl <<
                "\tretrieved data size: " << er.retrieved <<
                ", actual size: " << er.actual_size << endl;
        return -1;
    }
    catch (const Exception& er) {
        // Catch-all for any other MySQL++ exceptions
        cerr << "Error: " << er.what() << endl;
        return -1;
    }

    return 0;
}
```

Notice that we used the escape manipulator when building the INSERT query above. This is because mysqlpp::sql_blob is just an alias for one of the special MySQL++ string types, which don't do automatic quoting and escaping. They can't, because MySQL++ also uses these data types to hold raw SQL query strings, which would break due to doubled quoting and/or escaping if it were automatic.

## Serving images from BLOB column via CGI

The other example in this pair is rather short, considering how much it does. It parses a CGI query string giving the image ID, uses that to retreive data loaded into the database by `load_jpeg`, and writes it out in the form a web server wants when processing a CGI call, all with adequate real-world error handling. This is `examples/cgi_jpeg.cpp`:

```
#include "cmdline.h"
#include "images.h"

#define CRLF            "\r\n"
#define CRLF2           "\r\n\r\n"

int
main(int argc, char* argv[])
{
    // Get database access parameters from command line if present, else
    // use hard-coded values for true CGI case.
    mysqlpp::examples::CommandLine cmdline(argc, argv, "root",
            "nunyabinness");
    if (!cmdline) {
        return 1;
    }

    // Parse CGI query string environment variable to get image ID
    unsigned int img_id = 0;
    char* cgi_query = getenv("QUERY_STRING");
    if (cgi_query) {
        if ((strlen(cgi_query) < 4) || memcmp(cgi_query, "id=", 3)) {
            std::cout << "Content-type: text/plain" << std::endl << std::endl;
```

```
        std::cout << "ERROR: Bad query string" << std::endl;
        return 1;
    }
    else {
        img_id = atoi(cgi_query + 3);
    }
}
else {
    std::cerr << "Put this program into a web server's cgi-bin "
            "directory, then" << std::endl;
    std::cerr << "invoke it with a URL like this:" << std::endl;
    std::cerr << std::endl;
    std::cerr << "    http://server.name.com/cgi-bin/cgi_jpeg?id=2" <<
            std::endl;
    std::cerr << std::endl;
    std::cerr << "This will retrieve the image with ID 2." << std::endl;
    std::cerr << std::endl;
    std::cerr << "You will probably have to change some of the #defines "
            "at the top of" << std::endl;
    std::cerr << "examples/cgi_jpeg.cpp to allow the lookup to work." <<
            std::endl;
    return 1;
}

// Retrieve image from DB by ID
try {
    mysqlpp::Connection con(mysqlpp::examples::db_name,
            cmdline.server(), cmdline.user(), cmdline.pass());
    mysqlpp::Query query = con.query();
    query << "SELECT * FROM images WHERE id = " << img_id;
    mysqlpp::StoreQueryResult res = query.store();
    if (res && res.num_rows()) {
        images img = res[0];
        if (img.data.is_null) {
            std::cout << "Content-type: text/plain" << CRLF2;
            std::cout << "No image content!" << CRLF;
        }
        else {
            std::cout << "X-Image-Id: " << img_id << CRLF; // for debugging
            std::cout << "Content-type: image/jpeg" << CRLF;
            std::cout << "Content-length: " <<
                    img.data.data.length() << CRLF2;
            std::cout << img.data;
        }
    }
    else {
        std::cout << "Content-type: text/plain" << CRLF2;
        std::cout << "ERROR: No image with ID " << img_id << CRLF;
    }
}
catch (const mysqlpp::BadQuery& er) {
    // Handle any query errors
    std::cout << "Content-type: text/plain" << CRLF2;
    std::cout << "QUERY ERROR: " << er.what() << CRLF;
    return 1;
}
catch (const mysqlpp::Exception& er) {
    // Catch-all for any other MySQL++ exceptions
    std::cout << "Content-type: text/plain" << CRLF2;
    std::cout << "GENERAL ERROR: " << er.what() << CRLF;
    return 1;
}
```

```
    return 0;
}
```

While you can run it by hand, it's best to install this in a web server's CGI program directory, then call it with a URL like `http://my.server.com/cgi-bin/cgi_jpeg?id=1`. That retrieves the JPEG with ID 1 from the database and returns it to the web server, which will send it on to the browser.

We've included an image with MySQL++ that you can use with this example pair, `examples/logo.jpg`.

# 3.10. Using Transactions

The Transaction class makes it easier to use SQL transactions in an exception-safe manner. Normally you create the `Transaction` object on the stack before you issue the queries in your transaction set. Then, when all the queries in the transaction set have been issued, you call `Transaction::commit()`, which commits the transaction set. If the `Transaction` object goes out of scope before you call `commit()`, the transaction set is rolled back. This ensures that if some code throws an exception after the transaction is started but before it is committed, the transaction isn't left unresolved.

`examples/transaction.cpp` illustrates this:

```
#include "cmdline.h"
#include "printdata.h"
#include "stock.h"

#include <iostream>
#include <cstdio>

using namespace std;

int
main(int argc, char *argv[])
{
    // Get database access parameters from command line
    mysqlpp::examples::CommandLine cmdline(argc, argv);
    if (!cmdline) {
        return 1;
    }

    try {
        // Establish the connection to the database server.
        mysqlpp::Connection con(mysqlpp::examples::db_name,
                cmdline.server(), cmdline.user(), cmdline.pass());

        // Show initial state
        mysqlpp::Query query = con.query();
        cout << "Initial state of stock table:" << endl;
        print_stock_table(query);

        // Insert a few rows in a single transaction set
        {
            // Use a higher level of transaction isolation than MySQL
            // offers by default.  This trades some speed for more
            // predictable behavior.  We've set it to affect all
            // transactions started through this DB server connection,
            // so it affects the next block, too, even if we don't
            // commit this one.
            mysqlpp::Transaction trans(con,
                    mysqlpp::Transaction::serializable,
                    mysqlpp::Transaction::session);
```

```
            stock row("Sauerkraut", 42, 1.2, 0.75,
                    mysqlpp::sql_date("2006-03-06"), mysqlpp::null);
            query.insert(row);
            query.execute();

            cout << "\nRow inserted, but not committed." << endl;
            cout << "Verify this with another program (e.g. simple1), "
                    "then hit Enter." << endl;
            getchar();

            cout << "\nCommitting transaction gives us:" << endl;
            trans.commit();
            print_stock_table(query);
        }

        // Now let's test auto-rollback
        {
            // Start a new transaction, keeping the same isolation level
            // we set above, since it was set to affect the session.
            mysqlpp::Transaction trans(con);
            cout << "\nNow adding catsup to the database..." << endl;

            stock row("Catsup", 3, 3.9, 2.99,
                    mysqlpp::sql_date("2006-03-06"), mysqlpp::null);
            query.insert(row);
            query.execute();
        }
        cout << "\nNo, yuck! We don't like catsup. Rolling it back:" <<
                endl;
        print_stock_table(query);

    }
    catch (const mysqlpp::BadQuery& er) {
        // Handle any query errors
        cerr << "Query error: " << er.what() << endl;
        return -1;
    }
    catch (const mysqlpp::BadConversion& er) {
        // Handle bad conversions
        cerr << "Conversion error: " << er.what() << endl <<
                "\tretrieved data size: " << er.retrieved <<
                ", actual size: " << er.actual_size << endl;
        return -1;
    }
    catch (const mysqlpp::Exception& er) {
        // Catch-all for any other MySQL++ exceptions
        cerr << "Error: " << er.what() << endl;
        return -1;
    }

    return 0;
}
```

One of the downsides of transactions is that the locking it requires in the database server is prone to deadlocks. The classic case where this happens is when two programs both want access to the same two rows within a single transaction each, but they modify them in opposite orders. If the timing is such that the programs interleave their lock acquisitions, the two come to an impasse: neither can get access to the other row they want to modify until the other program commits its transaction and thus release the row locks, but neither can finish the transaction because they’re waiting on row locks the database server is holding on behalf of the other program.

The MySQL server is smart enough to detect this condition, but the best it can do is abort the second transaction. This breaks the impasse, allowing the first program to complete its transaction.

The second program now has to deal with the fact that its transaction just got aborted. There's a subtlety in detecting this situation when using MySQL++. By default, MySQL++ signals errors like these with exceptions. In the exception handler, you might expect to get `ER_LOCK_DEADLOCK` from `Query::errnum()` (or `Connection::errnum()`, same thing), but what you'll almost certainly get instead is 0, meaning "no error." Why? It's because you're probably using a `Transaction` object to get automatic roll-backs in the face of exceptions. In this case, the roll-back happens before your exception handler is called by issuing a **ROLLBACK** query to the database server. Thus, `Query::errnum()` returns the error code associated with this roll-back query, not the deadlocked transaction that caused the exception.

To avoid this problem, a few of the exception objects as of MySQL++ v3.0 include this last error number in the exception object itself. It's populated at the point of the exception, so it can differ from the value you would get from `Query::errnum()` later on when the exception handler runs.

The example `examples/deadlock.cpp` demonstrates the problem:

```
#include "cmdline.h"

#include <mysql++.h>
#include <mysqld_error.h>

#include <iostream>

using namespace std;

// Bring in global holding the value given to the -m switch
extern int run_mode;


int
main(int argc, char *argv[])
{
    // Get database access parameters from command line
    mysqlpp::examples::CommandLine cmdline(argc, argv);
    if (!cmdline) {
        return 1;
    }

    // Check that the mode parameter was also given and it makes sense
    const int run_mode = cmdline.run_mode();
    if ((run_mode != 1) && (run_mode != 2)) {
        cerr << argv[0] << " must be run with -m1 or -m2 as one of "
                "its command-line arguments." << endl;
        return 1;
    }

    mysqlpp::Connection con;
    try {
        // Establish the connection to the database server
        mysqlpp::Connection con(mysqlpp::examples::db_name,
                cmdline.server(), cmdline.user(), cmdline.pass());

        // Start a transaction set.  Transactions create mutex locks on
        // modified rows, so if two programs both touch the same pair of
        // rows but in opposite orders at the wrong time, one of the two
        // programs will deadlock.  The MySQL server knows how to detect
        // this situation, and its error return causes MySQL++ to throw
        // a BadQuery exception.  The point of this example is that if
        // you want to detect this problem, you would check the value of
        // BadQuery::errnum(), not Connection::errnum(), because the
        // transaction rollback process executes a query which succeeds,
        // setting the MySQL C API's "last error number" value to 0.
```

```
        // The exception object carries its own copy of the error number
        // at the point the exception was thrown for this very reason.
        mysqlpp::Query query = con.query();
        mysqlpp::Transaction trans(con);

        // Build and run the queries, with the order depending on the -m
        // flag, so that a second copy of the program will deadlock if
        // run while the first is waiting for Enter.
        char dummy[100];
        for (int i = 0; i < 2; ++i) {
            int lock = run_mode + (run_mode == 1 ? i : -i);
            cout << "Trying lock " << lock << "..." << endl;

            query << "select * from deadlock_test" << lock <<
                    " where x = " << lock << " for update";
            query.store();

            cout << "Acquired lock " << lock << ".  Press Enter to ";
            cout << (i == 0 ? "try next lock" : "exit");
            cout << ": " << flush;
            cin.getline(dummy, sizeof(dummy));
        }
    }
    catch (mysqlpp::BadQuery e) {
        if (e.errnum() == ER_LOCK_DEADLOCK) {
            cerr << "Transaction deadlock detected!" << endl;
            cerr << "Connection::errnum = " << con.errnum() <<
                    ", BadQuery::errnum = " << e.errnum() << endl;
        }
        else {
            cerr << "Unexpected query error: " << e.what() << endl;
        }
        return 1;
    }
    catch (mysqlpp::Exception e) {
        cerr << "General error: " << e.what() << endl;
        return 1;
    }

    return 0;
}
```

This example works a little differently than the others. You run one copy of the example, then when it pauses waiting for you to press **Enter**, you run another copy. Then, depending on which one you press **Enter** in, one of the two will abort with the deadlock exception. You can see from the error message you get that it matters which method you call to get the error number. What you do about it is up to you as it depends on your program's design and system architecture.

# 3.11. Which Query Type to Use?

There are three major ways to execute a query in MySQL++: `Query::execute()`, `Query::store()`, and `Query::use()`. Which should you use, and why?

`execute()` is for queries that do not return data *per se*. For instance, **CREATE INDEX**. You do get back some information from the MySQL server, which `execute()` returns to its caller in a SimpleResult object. In addition to the obvious — a flag stating whether the query succeeded or not — this object also contains things like the number of rows that the query affected. If you only need the success status, it's a little more efficient to call `Query::exec()` instead, as it simply returns bool.

If your query does pull data from the database, the simplest option is `store()`. (All of the examples up to this point have used this method.) This returns a StoreQueryResult object, which contains the entire result set. It's especially

convenient because `StoreQueryResult` derives from `std::vector<mysqlpp::Row>`, so it opens the whole panoply of STL operations for accessing the rows in the result set. Access rows randomly with subscript notation, iterate forwards and backwards over the result set, run STL algorithms on the set...it all works naturally.

If you like the idea of storing your results in an STL container but don't want to use `std::vector`, you can call `Query::storein()` instead. It lets you store the results in any standard STL container (yes, both sequential and set-associative types) instead of using `StoreQueryResult`. You do miss out on some of the additional database information held by `StoreQueryResult`'s other base class, ResultBase, however.

`store*()` queries are convenient, but the cost of keeping the entire result set in main memory can sometimes be too high. It can be surprisingly costly, in fact. A MySQL database server stores data compactly on disk, but it returns query data to the client in a textual form. This results in a kind of data bloat that affects numeric and BLOB types the most. MySQL++ and the underlying C API library also have their own memory overheads in addition to this. So, if you happen to know that the database server stores every record of a particular table in 1 KB, pulling a million records from that table could easily take several GB of memory with a `store()` query, depending on what's actually stored in that table.

For these large result sets, the superior option is a `use()` query. This returns a UseQueryResult object, which is similar to `StoreQueryResult`, but without all of the random-access features. This is because a "use" query tells the database server to send the results back one row at a time, to be processed linearly. It's analogous to a C++ stream's input iterator, as opposed to a random-access iterator that a container like vector offers. By accepting this limitation, you can process arbitrarily large result sets. This technique is demonstrated in `examples/simple3.cpp`:

```
#include "cmdline.h"
#include "printdata.h"

#include <mysql++.h>

#include <iostream>
#include <iomanip>

using namespace std;

int
main(int argc, char *argv[])
{
    // Get database access parameters from command line
    mysqlpp::examples::CommandLine cmdline(argc, argv);
    if (!cmdline) {
        return 1;
    }

    // Connect to the sample database.
    mysqlpp::Connection conn(false);
    if (conn.connect(mysqlpp::examples::db_name, cmdline.server(),
            cmdline.user(), cmdline.pass())) {
        // Ask for all rows from the sample stock table and display
        // them.  Unlike simple2 example, we retreive each row one at
        // a time instead of storing the entire result set in memory
        // and then iterating over it.
        mysqlpp::Query query = conn.query("select * from stock");
        if (mysqlpp::UseQueryResult res = query.use()) {
            // Display header
            cout.setf(ios::left);
            cout << setw(31) << "Item" <<
                    setw(10) << "Num" <<
                    setw(10) << "Weight" <<
                    setw(10) << "Price" <<
                    "Date" << endl << endl;
```

```
            // Get each row in result set, and print its contents
            while (mysqlpp::Row row = res.fetch_row()) {
                cout << setw(30) << row["item"] << ' ' <<
                        setw(9) << row["num"] << ' ' <<
                        setw(9) << row["weight"] << ' ' <<
                        setw(9) << row["price"] << ' ' <<
                        setw(9) << row["sdate"] <<
                        endl;
            }

            // Check for error: can't distinguish "end of results" and
            // error cases in return from fetch_row() otherwise.
            if (conn.errnum()) {
                cerr << "Error received in fetching a row: " <<
                        conn.error() << endl;
                return 1;
            }
            return 0;
        }
        else {
            cerr << "Failed to get stock item: " << query.error() << endl;
            return 1;
        }
    }
    else {
        cerr << "DB connection failed: " << conn.error() << endl;
        return 1;
    }
}
```

This example does the same thing as `simple2`, only with a “use” query instead of a “store” query.

Valuable as `use()` queries are, they should not be the first resort in solving problems of excessive memory use. It’s better if you can find a way to simply not pull as much data from the database in the first place. Maybe you’re saying **SELECT *** even though you don’t immediately need all the columns from the table. Or, maybe you’re filtering the result set with C++ code after you get it from the database server. If you can do that filtering with a more restrictive **WHERE** clause on the **SELECT**, it’ll not only save memory, it’ll save bandwidth between the database server and client, and can even save CPU time. If the filtering criteria can’t be expressed in a **WHERE** clause, however, read on to the next section.

# 3.12. Conditional Result Row Handling

Sometimes you must pull more data from the database server than you actually need and filter it in memory. SQL’s **WHERE** clause is powerful, but not as powerful as C++. Instead of storing the full result set and then picking over it to find the rows you want to keep, use `Query::store_if()`. This is `examples/store_if.cpp`:

```
#include "cmdline.h"
#include "printdata.h"
#include "stock.h"

#include <mysql++.h>

#include <iostream>

#include <math.h>


// Define a functor for testing primality.
struct is_prime
{
```

```
    bool operator()(const stock& s)
    {
        if ((s.num == 2) || (s.num == 3)) {
            return true;    // 2 and 3 are trivial cases
        }
        else if ((s.num < 2) || ((s.num % 2) == 0)) {
            return false;   // can't be prime if < 2 or even
        }
        else {
            // The only possibility left is that it's divisible by an
            // odd number that's less than or equal to its square root.
            for (int i = 3; i <= sqrt(double(s.num)); i += 2) {
                if ((s.num % i) == 0) {
                    return false;
                }
            }
            return true;
        }
    }
};


int
main(int argc, char *argv[])
{
    // Get database access parameters from command line
    mysqlpp::examples::CommandLine cmdline(argc, argv);
    if (!cmdline) {
        return 1;
    }

    try {
        // Establish the connection to the database server.
        mysqlpp::Connection con(mysqlpp::examples::db_name,
                cmdline.server(), cmdline.user(), cmdline.pass());

        // Collect the stock items with prime quantities
        std::vector<stock> results;
        mysqlpp::Query query = con.query();
        query.store_if(results, stock(), is_prime());

        // Show the results
        print_stock_header(results.size());
        std::vector<stock>::const_iterator it;
        for (it = results.begin(); it != results.end(); ++it) {
            print_stock_row(it->item.c_str(), it->num, it->weight,
                    it->price, it->sDate);
        }
    }
    catch (const mysqlpp::BadQuery& e) {
        // Something went wrong with the SQL query.
        std::cerr << "Query failed: " << e.what() << std::endl;
        return 1;
    }
    catch (const mysqlpp::Exception& er) {
        // Catch-all for any other MySQL++ exceptions
        std::cerr << "Error: " << er.what() << std::endl;
        return 1;
    }

    return 0;
}
```

I doubt anyone really needs to select rows from a table that have a prime number in a given field. This example is meant to be just barely more complex than SQL can manage, to avoid obscuring the point. That point being, the `Query::store_if()` call here gives you a container full of results meeting a criterion that you probably can't express in SQL. You will no doubt have much more useful criteria in your own programs.

If you need a more complex query than the one `store_if()` knows how to build when given an SSQLS examplar, there are two overloads that let you use your own query string. One overload takes the query string directly, and the other uses the query string built with `Query`'s stream interface.

# 3.13. Executing Code for Each Row In a Result Set

SQL is more than just a database query language. Modern database engines can actually do some calculations on the data on the server side. But, this isn't always the best way to get something done. When you need to mix code and a query, MySQL++'s `Query::for_each()` facility might be just what you need. This is `examples/for_each.cpp`:

```
#include "cmdline.h"
#include "printdata.h"
#include "stock.h"

#include <mysql++.h>

#include <iostream>

#include <math.h>


// Define a functor to collect statistics about the stock table
class gather_stock_stats
{
public:
    gather_stock_stats() :
    items_(0),
    weight_(0),
    cost_(0)
    {
    }

    void operator()(const stock& s)
    {
        items_  += s.num;
        weight_ += (s.num * s.weight);
        cost_   += (s.num * s.price.data);
    }

private:
    mysqlpp::sql_bigint items_;
    mysqlpp::sql_double weight_, cost_;

    friend std::ostream& operator<<(std::ostream& os,
            const gather_stock_stats& ss);
};


// Dump the contents of gather_stock_stats to a stream in human-readable
// form.
std::ostream&
operator<<(std::ostream& os, const gather_stock_stats& ss)
{
    os << ss.items_ << " items " <<
```

```
            "weighing " << ss.weight_ << " stone and " <<
            "costing " << ss.cost_ << " cowrie shells";
    return os;
}


int
main(int argc, char *argv[])
{
    // Get database access parameters from command line
    mysqlpp::examples::CommandLine cmdline(argc, argv);
    if (!cmdline) {
        return 1;
    }

    try {
        // Establish the connection to the database server.
        mysqlpp::Connection con(mysqlpp::examples::db_name,
                cmdline.server(), cmdline.user(), cmdline.pass());

        // Gather and display the stats for the entire stock table
        mysqlpp::Query query = con.query();
        std::cout << "There are " << query.for_each(stock(),
                gather_stock_stats()) << '.' << std::endl;
    }
    catch (const mysqlpp::BadQuery& e) {
        // Something went wrong with the SQL query.
        std::cerr << "Query failed: " << e.what() << std::endl;
        return 1;
    }
    catch (const mysqlpp::Exception& er) {
        // Catch-all for any other MySQL++ exceptions
        std::cerr << "Error: " << er.what() << std::endl;
        return 1;
    }

    return 0;
}
```

You only need to read the `main()` function to get a good idea of what the program does. The key line of code passes an SSQLS examplar and a functor to `Query::for_each()`. `for_each()` uses the SSQLS instance to build a `select * from TABLE` query, `stock` in this case. It runs that query internally, calling `gather_stock_stats` on each row. This is a pretty contrived example; you could actually do this in SQL, but we're trying to prevent the complexity of the code from getting in the way of the demonstration here.

Just as with `store_if()`, described above, there are two other overloads for `for_each()` that let you use your own query string.

# 3.14. Connection Options

MySQL has a large number of options that control how it makes the connection to the database server, and how that connection behaves. The defaults are sufficient for most programs, so only one of the MySQL++ example programs make any connection option changes. Here is `examples/multiquery.cpp`:

```
#include "cmdline.h"
#include "printdata.h"

#include <mysql++.h>

#include <algorithm>
```

```
#include <iostream>
#include <iomanip>
#include <vector>

using namespace std;
using namespace mysqlpp;


typedef vector<size_t> IntVectorType;


static void
print_header(IntVectorType& widths, StoreQueryResult& res)
{
    cout << "  |" << setfill(' ');
    for (size_t i = 0; i < res.field_names()->size(); i++) {
        cout << " " << setw(widths.at(i)) << res.field_name(int(i)) << " |";
    }
    cout << endl;
}


static void
print_row(IntVectorType& widths, Row& row)
{
    cout << "  |" << setfill(' ');
    for (size_t i = 0; i < row.size(); ++i) {
        cout << " " << setw(widths.at(i)) << row[int(i)] << " |";
    }
    cout << endl;
}


static void
print_row_separator(IntVectorType& widths)
{
    cout << "  +" << setfill('-');
    for (size_t i = 0; i < widths.size(); i++) {
        cout << "-" << setw(widths.at(i)) << '-' << "-+";
    }
    cout << endl;
}


static void
print_result(StoreQueryResult& res, int index)
{
    // Show how many rows are in result, if any
    StoreQueryResult::size_type num_results = res.size();
    if (res && (num_results > 0)) {
        cout << "Result set " << index << " has " << num_results <<
                " row" << (num_results == 1 ? "" : "s") << ':' << endl;
    }
    else {
        cout << "Result set " << index << " is empty." << endl;
        return;
    }

    // Figure out the widths of the result set's columns
    IntVectorType widths;
    size_t size = res.num_fields();
    for (size_t i = 0; i < size; i++) {
        widths.push_back(max(
```

```
                res.field(i).max_length(),
                res.field_name(i).size()));
    }

    // Print result set header
    print_row_separator(widths);
    print_header(widths, res);
    print_row_separator(widths);

    // Display the result set contents
    for (StoreQueryResult::size_type i = 0; i < num_results; ++i) {
        print_row(widths, res[i]);
    }

    // Print result set footer
    print_row_separator(widths);
}


static void
print_multiple_results(Query& query)
{
    // Execute query and print all result sets
    StoreQueryResult res = query.store();
    print_result(res, 0);
    for (int i = 1; query.more_results(); ++i) {
        res = query.store_next();
        print_result(res, i);
    }
}


int
main(int argc, char *argv[])
{
    // Get connection parameters from command line
    mysqlpp::examples::CommandLine cmdline(argc, argv);
    if (!cmdline) {
        return 1;
    }

    try {
        // Enable multi-queries.  Notice that you almost always set
        // MySQL++ connection options before establishing the server
        // connection, and options are always set using this one
        // interface.  If you're familiar with the underlying C API,
        // you know that there is poor consistency on these matters;
        // MySQL++ abstracts these differences away.
        Connection con;
        con.set_option(new MultiStatementsOption(true));

        // Connect to the database
        if (!con.connect(mysqlpp::examples::db_name, cmdline.server(),
                cmdline.user(), cmdline.pass())) {
            return 1;
        }

        // Set up query with multiple queries.
        Query query = con.query();
        query << "DROP TABLE IF EXISTS test_table; " <<
                "CREATE TABLE test_table(id INT); " <<
                "INSERT INTO test_table VALUES(10); " <<
                "UPDATE test_table SET id=20 WHERE id=10; " <<
```

```
                "SELECT * FROM test_table; " <<
                "DROP TABLE test_table";
        cout << "Multi-query: " << endl << query << endl;

        // Execute statement and display all result sets.
        print_multiple_results(query);

#if MYSQL_VERSION_ID >= 50000
        // If it's MySQL v5.0 or higher, also test stored procedures, which
        // return their results the same way multi-queries do.
        query << "DROP PROCEDURE IF EXISTS get_stock; " <<
                "CREATE PROCEDURE get_stock" <<
                "( i_item varchar(20) ) " <<
                "BEGIN " <<
                "SET i_item = concat('%', i_item, '%'); " <<
                "SELECT * FROM stock WHERE lower(item) like lower(i_item); " <<
                "END;";
        cout << "Stored procedure query: " << endl << query << endl;

        // Create the stored procedure.
        print_multiple_results(query);

        // Call the stored procedure and display its results.
        query << "CALL get_stock('relish')";
        cout << "Query: " << query << endl;
        print_multiple_results(query);
#endif

        return 0;
    }
    catch (const BadOption& err) {
        cerr << err.what() << endl;
        cerr << "This example requires MySQL 4.1.1 or later." << endl;
        return 1;
    }
    catch (const ConnectionFailed& err) {
        cerr << "Failed to connect to database server: " <<
                err.what() << endl;
        return 1;
    }
    catch (const Exception& er) {
        // Catch-all for any other MySQL++ exceptions
        cerr << "Error: " << er.what() << endl;
        return 1;
    }
}
```

This is a fairly complex example demonstrating the multi-query and stored procedure features in newer versions of MySQL. Because these are new features, and they change the communication between the client and server, you have to enable these features in a connection option. The key line is right up at the top of `main()`, where it creates a MultiStatementsOption object and passes it to `Connection::set_option()`. That method will take a pointer to any derivative of Option: you just create such an object on the heap and pass it in, which gives `Connection` the data values it needs to set the option. You don't need to worry about releasing the memory used by the `Option` objects; it's done automatically.

The only tricky thing about setting options is that only a few of them can be set after the connection is up. Most need to be set just as shown in the example above: create an unconnected `Connection` object, set your connection options, and only then establish the connection. The option setting mechanism takes care of applying the options at the correct time in the connection establishment sequence.

If you're familiar with setting connection options in the MySQL C API, you'll have to get your head around the fact that MySQL++'s connection option mechanism is a much simpler, higher-level design that doesn't resemble the C API in any way. The C API has something like half a dozen different mechanisms for setting options that control the connection. The flexibility of the C++ type system allows us to wrap all of these up into a single high-level mechanism while actually getting greater type safety than the C API allows.

# 3.15. Dealing with Connection Timeouts

By default, current MySQL servers have an 8 hour idle timeout on connections. This is not a problem if your program never has to run for more than 8 hours or reliably queries the database more often than that. And, it's a good thing for the database server, because even an idle connection takes up server resources.

Many programs must run continually, however, and may experience long idle periods, such as nights and weekends when no one is around to make the program issue database queries. It's therefore common for people writing such programs to get a bug report from the field complaining that the program died overnight or over a long weekend, usually with some error message about the database server going away. They then check the DB server, find that it's still running and never did restart and scratch their heads wondering what happened. What happened is that the server's connection idle timeout expired, so it closed the connection to the client.

You cannot detect this condition by calling `Connection::connected()`. When that returns true, it just means that either the connect-on-create constructor or the `connect()` call succeeded and that we haven't observed the connection to be down since then. When the database server closes an idle connection, you won't know it until after you try to issue a query. This is simply due to the nature of network programming.

One way around this problem is to configure MySQL to have a longer idle timeout. This timeout is in seconds, so the default of 8 hours is 28,800 seconds. You would want to figure out the longest possible time that your program could be left idle, then pick a value somewhat longer than that. For instance, you might decide that the longest reasonable idle time is a long 4-day weekend — 345,600 seconds — which you could round up to 350,000 or 400,000 to allow for a little bit of additional idle time on either end of that period.

Another way around this, on a per-connection basis from the client side, would be to set the ReconnectOption connection option. This will cause MySQL++ to reconnect to the server automatically if it drops the connection. Beware that unless you're using MySQL 5.1.6 or higher, you have to set this only after the connection is established, or it won't take effect. This means there's a potential race condition: it's possible the connection could drop shortly enough after being established that you don't have time to apply the option, so it won't come back up automatically. MySQL 5.1.6+ fixes this by allowing this option to be set before the connection is established.

A completely different way to tackle this, if your program doesn't block forever waiting on I/O while idle, is to periodically call `Connection::ping()`.[12] This sends the smallest possible amount of data to the database server, which will reset its idle timer and cause it to respond, so `ping()` returns true. If it returns false instead, you know you need to reconnect to the server. Periodic pinging is easiest to do if your program uses asynchronous I/O, threads, or some kind of event loop to ensure that you can call something periodically even while the rest of the program has nothing to do.

An interesting variant on this strategy is to ping the server before each query, or, better, before each group of queries within a larger operation. It has an advantage over pinging during idle time in that the client is about to use far more server resources to handle the query than it will take to handle the ping, so the ping time gets lost in the overhead. On the other hand, if the client issues queries frequently when not idle, it can result in a lot more pings than would happen if you just pinged every N hours while idle.

[12] Don't ping the server too often! It takes a tiny amount of processing capability to handle a ping, which can add up to a significant amount if done often enough by a client, or even just rarely by enough clients. Also, a lower ping frequency can let your program ride through some types of network faults — a switch reboot, for instance — without needing a reconnect. I like to ping the DB server no more often than half the connection timeout. With the default of 8 hours, then, I'd ping between every 4 and 7 hours.

Finally, some programmers prefer to wrap the querying mechanism in an error handler that catches the “server has gone away” error and tries to reestablish the connection and reissue the query. This adds some complexity, but it makes your program more robust without taking up unnecessary resources. If you did this, you could even change the server to drop idle connections more often, thus tying up fewer TCP/IP stack resources.

# 3.16. Concurrent Queries on a Connection

An important limitation of the MySQL C API library — which MySQL++ is built atop, so it shares this limitation — is that you can only have one query in progress on each connection to the database server. If you try to issue a second query while one is still in progress, you get an obscure error message about “Commands out of sync” from the underlying C API library. (You normally get this message in a MySQL++ exception unless you have exceptions disabled, in which case you get a failure code and `Connection::error()` returns this message.)

There are lots of ways to run into this limitation:

- The easiest way is to try to use a single Connection object in a multithreaded program, with more than one thread attempting to use it to issue queries. Unless you put in a lot of work to synchronize access, this is almost guaranteed to fail at some point, giving the dread “Commands out of sync” error.

- You might then think to give each thread that issues queries its own `Connection` object. You can still run into trouble if you pass the data you get from queries around to other threads. What can happen is that one of these child objects indirectly calls back to the `Connection` at a time where it’s involved with another query. This is properly covered elsewhere, in Section 7.4, “Sharing MySQL++ Data Structures”.)

- One way to run into this problem without using threads is with “use” queries, discussed above. If you don’t consume all rows from a query before you issue another on that connection, you are effectively trying to have multiple concurrent queries on a single connection. Here’s a recipie for this particular disaster:

  ```
  UseQueryResult r1 = query.use("select garbage from plink where foobie='tamagotchi'");
  UseQueryResult r2 = query.use("select blah from bonk where bletch='smurf'");
  ```

  The second `use()` call fails because the first result set hasn’t been consumed yet.

- Still another way to run into this limitation is if you use MySQL’s multi-query feature. This lets you give multiple queries in a single call, separated by semicolons, and get back the results for each query separately. If you issue three queries using `Query::store()`, you only get back the first query’s results with that call, and then have to call `store_next()` to get the subsequent query results. MySQL++ provides `Query::more_results()` so you know whether you’re done, or need to call `store_next()` again. Until you reach the last result set, you can’t issue another query on that connection.

- Finally, there’s a way to run into this that surprises almost everyone sooner or later: stored procedures. MySQL normally returns *at least two* result sets for a stored procedure call. The simple case is that the stored procedure contains a single SQL query, and it succeeds: you get two results, first the results of the embedded SQL query, and then the result of the call itself. If there are multiple SQL queries within the stored procedure, you get more than two result sets. Until you consume them all, you can’t start a new query on the connection. As above, you want to have a loop calling `more_results()` and `store_next()` to work your way through all of the result sets produced by the stored procedure call.

# 3.17. Getting Field Meta-Information

The following example demonstrates how to get information about the fields in a result set, such as the name of the field and the SQL type. This is `examples/fieldinf.cpp`:

```
#include "cmdline.h"
#include "printdata.h"

#include <iostream>
#include <iomanip>

using namespace std;


int
main(int argc, char *argv[])
{
    // Get database access parameters from command line
    mysqlpp::examples::CommandLine cmdline(argc, argv);
    if (!cmdline) {
        return 1;
    }

    try {
        // Establish the connection to the database server.
        mysqlpp::Connection con(mysqlpp::examples::db_name,
                cmdline.server(), cmdline.user(), cmdline.pass());

        // Get contents of main example table
        mysqlpp::Query query = con.query("select * from stock");
        mysqlpp::StoreQueryResult res = query.store();

        // Show info about each field in that table
        char widths[] = { 12, 22, 46 };
        cout.setf(ios::left);
        cout << setw(widths[0]) << "Field" <<
                setw(widths[1]) << "SQL Type" <<
                setw(widths[2]) << "Equivalent C++ Type" <<
                endl;
        for (size_t i = 0; i < sizeof(widths) / sizeof(widths[0]); ++i) {
            cout << string(widths[i] - 1, '=') << ' ';
        }
        cout << endl;

        for (size_t i = 0; i < res.field_names()->size(); i++) {
            // Suppress C++ type name outputs when run under dtest,
            // as they're system-specific.
            const char* cname = res.field_type(int(i)).name();
            mysqlpp::FieldTypes::value_type ft = res.field_type(int(i));
            ostringstream os;
            os << ft.sql_name() << " (" << ft.id() << ')';
            cout << setw(widths[0]) << res.field_name(int(i)).c_str() <<
                    setw(widths[1]) << os.str() <<
                    setw(widths[2]) << cname <<
                    endl;
        }
        cout << endl;

        // Simple type check
        if (res.field_type(0) == typeid(string)) {
            cout << "SQL type of 'item' field most closely resembles "
                    "the C++ string type." << endl;
        }

        // Tricky type check: the 'if' path shouldn't happen because the
        // description field has the NULL attribute.  We need to dig a
        // little deeper if we want to ignore this in our type checks.
        if (res.field_type(5) == typeid(string)) {
```

```
            cout << "Should not happen! Type check failure." << endl;
        }
        else if (res.field_type(5) == typeid(mysqlpp::sql_blob_null)) {
            cout << "SQL type of 'description' field resembles "
                    "a nullable variant of the C++ string type." << endl;
        }
        else {
            cout << "Weird: fifth field's type is now " <<
                    res.field_type(5).name() << endl;
            cout << "Did something recently change in resetdb?" << endl;
        }
    }
    catch (const mysqlpp::BadQuery& er) {
        // Handle any query errors
        cerr << "Query error: " << er.what() << endl;
        return -1;
    }
    catch (const mysqlpp::Exception& er) {
        // Catch-all for any other MySQL++ exceptions
        cerr << "Error: " << er.what() << endl;
        return -1;
    }

    return 0;
}
```

# 4. Template Queries

Another powerful feature of MySQL++ is being able to set up template queries. These are kind of like C's `printf()` facility: you give MySQL++ a string containing the fixed parts of the query and placeholders for the variable parts, and you can later substitute in values into those placeholders.

The following program demonstrates how to use this feature. This is `examples/tquery1.cpp`:

```
#include "cmdline.h"
#include "printdata.h"

#include <iostream>

using namespace std;

int
main(int argc, char *argv[])
{
    // Get database access parameters from command line
    mysqlpp::examples::CommandLine cmdline(argc, argv);
    if (!cmdline) {
        return 1;
    }

    try {
        // Establish the connection to the database server.
        mysqlpp::Connection con(mysqlpp::examples::db_name,
                cmdline.server(), cmdline.user(), cmdline.pass());

        // Build a template query to retrieve a stock item given by
        // item name.
        mysqlpp::Query query = con.query(
                "select * from stock where item = %0q");
        query.parse();

        // Retrieve an item added by resetdb; it won't be there if
        // tquery* or ssqls3 is run since resetdb.
        mysqlpp::StoreQueryResult res1 = query.store("Nürnberger Brats");
        if (res1.empty()) {
            throw mysqlpp::BadQuery("UTF-8 bratwurst item not found in "
                    "table, run resetdb");
        }

        // Replace the proper German name with a 7-bit ASCII
        // approximation using a different template query.
        query.reset();      // forget previous template query data
        query << "update stock set item = %0q where item = %1q";
        query.parse();
        mysqlpp::SimpleResult res2 = query.execute("Nuerenberger Bratwurst",
                res1[0][0].c_str());

        // Print the new table contents.
        print_stock_table(query);
    }
    catch (const mysqlpp::BadQuery& er) {
        // Handle any query errors
        cerr << "Query error: " << er.what() << endl;
        return -1;
    }
    catch (const mysqlpp::BadConversion& er) {
        // Handle bad conversions
```

```
        cerr << "Conversion error: " << er.what() << endl <<
                "\tretrieved data size: " << er.retrieved <<
                ", actual size: " << er.actual_size << endl;
        return -1;
    }
    catch (const mysqlpp::Exception& er) {
        // Catch-all for any other MySQL++ exceptions
        cerr << "Error: " << er.what() << endl;
        return -1;
    }

    return 0;
}
```

The line just before the call to `query.parse()` sets the template, and the parse call puts it into effect. From that point on, you can re-use this query by calling any of several Query member functions that accept query template parameters. In this example, we're using `Query::execute()`.

Let's dig into this feature a little deeper.

# 4.1. Setting up Template Queries

To set up a template query, you simply insert it into the Query object, using numbered placeholders wherever you want to be able to change the query. Then, you call the parse() function to tell the Query object that the query string is a template query, and it needs to parse it:

```
query << "select (%2:field1, %3:field2) from stock where %1:wheref = %0q:what";
query.parse();
```

The format of the placeholder is:

```
%###(modifier)(:name)(:)
```

Where "###" is a number up to three digits. It is the order of parameters given to a SQLQueryParms object, starting from 0.

"modifier" can be any one of the following:

| | |
|---|---|
| **%** | Print an actual "%" |
| "" | Don't quote or escape no matter what. |
| **q** | This will escape the item using the MySQL C API function mysql-escape-string and add single quotes around it as necessary, depending on the type of the value you use. |
| **Q** | Quote but don't escape based on the same rules as for "q". This can save a bit of processing time if you know the strings will never need quoting |

":name" is for an optional name which aids in filling SQLQueryParms. Name can contain any alpha-numeric characters or the underscore. You can have a trailing colon, which will be ignored. If you need to represent an actual colon after the name, follow the name with two colons. The first one will end the name and the second one won't be processed.

## 4.2. Setting the Parameters at Execution Time

To specify the parameters when you want to execute a query simply use `Query::store(const SQLString &parm0, [..., const SQLString &parm11])`. This type of multiple overload also exists for `Query::storein()`, `Query::use()` and `Query::execute()`. "parm0" corresponds to the first parameter, etc. You may specify up to 25 parameters. For example:

```
StoreQueryResult res = query.store("Dinner Rolls", "item", "item", "price")
```

with the template query provided above would produce:

```
select (item, price) from stock where item = "Dinner Rolls"
```

The reason we didn't put the template parameters in numeric order...

```
select (%0:field1, %1:field2) from stock where %2:wheref = %3q:what
```

...will become apparent shortly.

## 4.3. Default Parameters

The template query mechanism allows you to set default parameter values. You simply assign a value for the parameter to the appropriate position in the `Query::template_defaults` array. You can refer to the parameters either by position or by name:

```
query.template_defaults[1] = "item";
query.template_defaults["wheref"] = "item";
```

Both do the same thing.

This mechanism works much like C++'s default function parameter mechanism: if you set defaults for the parameters at the end of the list, you can call one of `Query`'s query execution methods without passing all of the values. If the query takes four parameters and you've set defaults for the last three, you can execute the query using as little as just one explicit parameter.

Now you can see why we numbered the template query parameters the way we did a few sections earlier. We ordered them so that the ones less likely to change have higher numbers, so we don't always have to pass them. We can just give them defaults and take those defaults when applicable. This is most useful when some parameters in a template query vary less often than other parameters. For example:

```
query.template_defaults["field1"] = "item";
query.template_defaults["field2"] = "price";
StoreQueryResult res1 = query.store("Hamburger Buns", "item");
StoreQueryResult res2 = query.store(1.25, "price");
```

This stores the result of the following queries in `res1` and `res2`, respectively:

```
select (item, price) from stock where item = "Hamburger Buns"
select (item, price) from stock where price = 1.25
```

Default parameters are useful in this example because we have two queries to issue, and parameters 2 and 3 remain the same for both, while parameters 0 and 1 vary.

Some have been tempted into using this mechanism as a way to set all of the template parameters in a query:

```
query.template_defaults["what"] = "Hamburger Buns";
query.template_defaults["wheref"] = "item";
query.template_defaults["field1"] = "item";
query.template_defaults["field2"] = "price";
StoreQueryResult res1 = query.store();
```

This can work, but it is *not designed to*. In fact, it's known to fail horribly in one common case. You will not get sympathy if you complain on the mailing list about it not working. If your code doesn't actively reuse at least one of the parameters in subsequent queries, you're abusing MySQL++, and it is likely to take its revenge on you.

# 4.4. Error Handling

If for some reason you did not specify all the parameters when executing the query and the remaining parameters do not have their values set via `Query::template_defaults`, the query object will throw a BadParamCount object. If this happens, you can get an explanation of what happened by calling `BadParamCount::what()`, like so:

```
query.template_defaults["field1"] = "item";
query.template_defaults["field2"] = "price";
StoreQueryResult res = query.store(1.25);
```

This would throw `BadParamCount` because the `wheref` is not specified.

In theory, this exception should never be thrown. If the exception is thrown it probably a logic error in your program.

# 5. Specialized SQL Structures

The Specialized SQL Structure (SSQLS) feature lets you easily define C++ structures that match the form of your SQL tables. At the most superficial level, an SSQLS has a member variable corresponding to each field in the SQL table. But, an SSQLS also has several methods, operators, and data members used by MySQL++'s internals to provide neat functionality, which we cover in this chapter.

You define SSQLSes using the macros defined in `ssqls.h`. This is the only MySQL++ header not automatically included for you by `mysql++.h`. You have to include it in code modules that use the SSQLS feature.

## 5.1. sql_create

Let's say you have the following SQL table:

```
CREATE TABLE stock (
    item CHAR(30) NOT NULL,
    num BIGINT NOT NULL,
    weight DOUBLE NOT NULL,
    price DECIMAL(6,2) NOT NULL,
    sdate DATE NOT NULL,
    description MEDIUMTEXT NULL)
```

You can create a C++ structure corresponding to this table like so:

```
sql_create_6(stock, 1, 6,
    mysqlpp::sql_char, item,
    mysqlpp::sql_bigint, num,
    mysqlpp::sql_double, weight,
    mysqlpp::sql_decimal, price,
    mysqlpp::sql_date, sdate,
    mysqlpp::Null<mysqlpp::sql_mediumtext>, description)
```

This declares the `stock` structure, which has a data member for each SQL column, using the same names. The structure also has a number of member functions, operators and hidden data members, but we won't go into that just now.

The parameter before each field name in the `sql_create_#` call is the C++ data type that will be used to hold that value in the SSQLS. While you could use plain old C++ data types for most of these columns (long int instead of mysqlpp::sql_bigint, for example) it's best to use the MySQL++ typedefs.

Sometimes you have no choice but to use special MySQL++ data types to fully express the database schema. Consider the `description` field. MySQL++'s sql_mediumtext type is just an alias for std::string, since we don't need anything fancier to hold a SQL MEDIUMTEXT value. It's the SQL NULL attribute that causes trouble: it has no equivalent in the C++ type system. MySQL++ offers the Null template, which bridges this difference between the two type systems.

The general format of this macro is:

```
sql_create_#(NAME, COMPCOUNT, SETCOUNT, TYPE1, ITEM1, ... TYPE#, ITEM#)
```

where # is the number of member variables, *`NAME`* is the name of the structure you wish to create, *`TYPEx`* is the type of a member variable, and *`ITEMx`* is that variable's name.

The *`COMPCOUNT`* and *`SETCOUNT`* arguments are described in the next section.

# 5.2. SSQLS Comparison and Initialization

The `sql_create_#` macro adds member functions and operators to each SSQLS that allow you to compare one SSQLS instance to another. These functions compare the first *`COMPCOUNT`* fields in the structure. In the example above, *`COMPCOUNT`* is 1, so only the `item` field will be checked when comparing two `stock` structures.

This feature works best when your table's "key" fields are the first ones in the SSQLS and you set *`COMPCOUNT`* equal to the number of key fields. That way, a check for equality between two SSQLS structures in your C++ code will give the same results as a check for equality in SQL.

*`COMPCOUNT`* must be at least 1. The current implementation of `sql_create_#` cannot create an SSQLS without comparison member functions.

Because our `stock` structure is less-than-comparable, you can use it in STL algorithms and containers that require this, such as STL's associative containers:

```
std::set<stock> result;
query.storein(result);
cout << result.lower_bound(stock("Hamburger"))->item << endl;
```

This will print the first item in the result set that begins with "Hamburger."

The third parameter to `sql_create_#` is *`SETCOUNT`*. If this is nonzero, it adds an initialization constructor and a `set()` member function taking the given number of arguments, for setting the first *N* fields of the structure. For example, you could change the above example like so:

```
sql_create_6(stock, 1, 2,
    mysqlpp::sql_char, item,
    mysqlpp::sql_bigint, num,
    mysqlpp::sql_double, weight,
    mysqlpp::sql_decimal, price,
    mysqlpp::sql_date, sdate,
    mysqlpp::Null<mysqlpp::sql_mediumtext>, description)

stock foo("Hotdog", 52);
```

In addition to this 2-parameter constructor, this version of the `stock` SSQLS will have a similar 2-parameter `set()` member function.

The *`COMPCOUNT`* and *`SETCOUNT`* values cannot be equal. If they are, the macro will generate two initialization constructors with identical parameter lists, which is illegal in C++. You might be asking, why does there need to be a constructor for comparison to begin with? It's often convenient to be able to say something like **`x == stock("Hotdog")`**. This requires that there be a constructor taking *`COMPCOUNT`* arguments to create the temporary `stock` instance used in the comparison.

This limitation is not a problem in practice. If you want the same number of parameters in the initialization constructor as the number of fields used in comparisons, pass 0 for *`SETCOUNT`*. This suppresses the duplicate constructor you'd get if you used the *`COMPCOUNT`* value instead. This is most useful in very small SSQLSes, since it's easier for the number of key fields to equal the number of fields you want to compare on:

```
sql_create_1(stock_item, 1, 0, mysqlpp::sql_char, item)
```

# 5.3. Retrieving data

Let's put SSQLS to use. This is `examples/ssqls1.cpp`:

```
#include "cmdline.h"
#include "printdata.h"
#include "stock.h"

#include <iostream>
#include <vector>

using namespace std;

int
main(int argc, char *argv[])
{
    // Get database access parameters from command line
    mysqlpp::examples::CommandLine cmdline(argc, argv);
    if (!cmdline) {
        return 1;
    }

    try {
        // Establish the connection to the database server.
        mysqlpp::Connection con(mysqlpp::examples::db_name,
                cmdline.server(), cmdline.user(), cmdline.pass());

        // Retrieve a subset of the stock table's columns, and store
        // the data in a vector of 'stock' SSQLS structures.  See the
        // user manual for the consequences arising from this quiet
        // ability to store a subset of the table in the stock SSQLS.
        mysqlpp::Query query = con.query("select item,description from stock");
        vector<stock> res;
        query.storein(res);

        // Display the items
        cout << "We have:" << endl;
        vector<stock>::iterator it;
        for (it = res.begin(); it != res.end(); ++it) {
            cout << '\t' << it->item;
            if (it->description != mysqlpp::null) {
                cout << " (" << it->description << ")";
            }
            cout << endl;
        }
    }
    catch (const mysqlpp::BadQuery& er) {
        // Handle any query errors
        cerr << "Query error: " << er.what() << endl;
        return -1;
    }
    catch (const mysqlpp::BadConversion& er) {
        // Handle bad conversions; e.g. type mismatch populating 'stock'
        cerr << "Conversion error: " << er.what() << endl <<
                "\tretrieved data size: " << er.retrieved <<
                ", actual size: " << er.actual_size << endl;
        return -1;
    }
    catch (const mysqlpp::Exception& er) {
        // Catch-all for any other MySQL++ exceptions
        cerr << "Error: " << er.what() << endl;
        return -1;
```

```
    }

    return 0;
}
```

Here is the stock.h header used by that example, and by several others below:

```
#include <mysql++.h>
#include <ssqls.h>

// The following is calling a very complex macro which will create
// "struct stock", which has the member variables:
//
//   sql_char item;
//   ...
//   sql_mediumtext_null description;
//
// plus methods to help populate the class from a MySQL row.  See the
// SSQLS sections in the user manual for further details.
sql_create_6(stock,
    1, 6, // The meaning of these values is covered in the user manual
    mysqlpp::sql_char, item,
    mysqlpp::sql_bigint, num,
    mysqlpp::sql_double, weight,
    mysqlpp::sql_double_null, price,
    mysqlpp::sql_date, sDate,           // SSQLS isn't case-sensitive!
    mysqlpp::sql_mediumtext_null, description)
```

This example produces the same output as `simple1.cpp` (see Section 3.2, “A Simple Example”), but it uses higher-level data structures paralleling the database schema instead of MySQL++’s lower-level generic data structures. It also uses MySQL++’s exceptions for error handling instead of doing everything inline. For small example programs like these, the overhead of SSQLS and exceptions doesn’t pay off very well, but in a real program, they end up working much better than hand-rolled code.

Notice that we are only pulling a single column from the `stock` table, but we are storing the rows in a std::vector<stock>. It may strike you as inefficient to have five unused fields per record. It’s easily remedied by defining a subset SSQLS:

```
sql_create_1(stock_subset,
  1, 0,
  string, item)

vector<stock_subset> res;
query.storein(res);
// ...etc...
```

MySQL++ is flexible about populating SSQLSes.[13] It works much like the Web, a design that’s enabled the development of the largest distributed system in the world. Just as a browser ignores tags and attributes it doesn’t understand, you can populate an SSQLS from a query result set containing columns that don’t exist in the SSQLS. And as a browser uses sensible defaults when the page doesn’t give explicit values, you can have an SSQLS with more fields defined than are in the query result set, and these SSQLS fields will get default values. (Zero for numeric types, false for bool, and a type-specific default for anything more complex, like mysqlpp::DateTime.)

[13]Programs built against versions of MySQL++ prior to 3.0 would crash at almost any mismatch between the database schema and the SSQLS definition. It’s no longer necessary to keep the data design in lock-step between the client and database server. A mismatch can result in data loss, but not a crash.

In more concrete terms, the example above is able to populate the `stock` objects using as much information as it has, and leave the remaining fields at their defaults. Conversely, you could also stuff the results of `SELECT * FROM stock` into the `stock_subset` SSQLS declared above; the extra fields would just be ignored.

We're trading run-time efficiency for flexibility here, usually the right thing in a distributed system. Since MySQL is a networked database server, many uses of it will qualify as distributed systems. You can't count on being able to update both the server(s) and all the clients at the same time, so you have to make them flexible enough to cope with differences while the changes propagate. As long as the new database schema isn't too grossly different from the old, your programs should continue to run until you get around to updating them to use the new schema.

There's a danger that this quiet coping behavior may mask problems, but considering that the previous behavior was for the program to crash when the database schema got out of synch with the SSQLS definition, it's likely to be taken as an improvement.

# 5.4. Adding data

MySQL++ offers several ways to insert data in SSQLS form into a database table.

## Inserting a Single Row

The simplest option is to insert a single row at a time. This is `examples/ssqls2.cpp`:

```
#include "cmdline.h"
#include "printdata.h"
#include "stock.h"

#include <iostream>
#include <limits>

using namespace std;

int
main(int argc, char *argv[])
{
    // Get database access parameters from command line
    mysqlpp::examples::CommandLine cmdline(argc, argv);
    if (!cmdline) {
        return 1;
    }

    try {
        // Establish the connection to the database server.
        mysqlpp::Connection con(mysqlpp::examples::db_name,
                cmdline.server(), cmdline.user(), cmdline.pass());

        // Create and populate a stock object.  We could also have used
        // the set() member, which takes the same parameters as this
        // constructor.
        stock row("Hot Dogs", 100, 1.5,
                numeric_limits<double>::infinity(), // "priceless," ha!
                mysqlpp::sql_date("1998-09-25"), mysqlpp::null);

        // Form the query to insert the row into the stock table.
        mysqlpp::Query query = con.query();
        query.insert(row);

        // Show the query about to be executed.
        cout << "Query: " << query << endl;
```

```
        // Execute the query.  We use execute() because INSERT doesn't
        // return a result set.
        query.execute();

        // Retrieve and print out the new table contents.
        print_stock_table(query);
    }
    catch (const mysqlpp::BadQuery& er) {
        // Handle any query errors
        cerr << "Query error: " << er.what() << endl;
        return -1;
    }
    catch (const mysqlpp::BadConversion& er) {
        // Handle bad conversions
        cerr << "Conversion error: " << er.what() << endl <<
                "\tretrieved data size: " << er.retrieved <<
                ", actual size: " << er.actual_size << endl;
        return -1;
    }
    catch (const mysqlpp::Exception& er) {
        // Catch-all for any other MySQL++ exceptions
        cerr << "Error: " << er.what() << endl;
        return -1;
    }

    return 0;
}
```

That's all there is to it! MySQL++ even takes care of quoting and escaping the data when building queries from SSQLS structures. It's efficient, too: MySQL++ is smart enough to quote and escape data only for those data types that actually require it.

## Inserting Many Rows

Inserting a single row is useful, to be sure, but you might want to be able to insert many SSQLSes or `Row` objects at once. MySQL++ knows how to do that, too, sparing you the necessity of writing the loop. Plus, MySQL++ uses an optimized implementation of this algorithm, packing everything into a single SQL query, eliminating the overhead of multiple calls between the client and server. It's just a different overload of `insert()`, which accepts a pair of iterators into an STL container, inserting every row in that range:

```
vector<stock> lots_of_stuff;
...populate the vector somehow...
query.insert(lots_of_stuff.begin(), lots_of_stuff.end()).execute();
```

By the way, notice that you can chain `Query` operations like in the last line above, because its methods return *this where that makes sense.

## Working Around MySQL's Packet Size Limit

The two-iterator form of `insert()` has an associated risk: MySQL has a limit on the size of the SQL query it will process. The default limit is 1 MB. You can raise the limit, but the reason the limit is configurable is not to allow huge numbers of inserts in a single query. They made the limit configurable because a single row might be bigger than 1 MB, so the default would prevent you from inserting anything at all. If you raise the limit simply to be able to insert more rows at once, you're courting disaster with no compensating benefit: the more data you send at a time, the greater the chance and cost of something going wrong. Worse, this is pure risk, because by the time you hit 1 MB, the per-packet overhead is such a small fraction of the data being transferred that increasing the packet size buys you essentially nothing.

Let's say you have a `vector` containing several megabytes of data; it will get even bigger when expressed in SQL form, so there's no way you can insert it all in a single query without raising the MySQL packet limit. One way to cope would be to write your own naïve loop, inserting just one row at a time. This is slow, because you're paying the per-query cost for every row in the container. Then you might realize that you could use the two iterator form of `insert()`, passing iterators expressing sub-ranges of the container instead of trying to insert the whole container in one go. Now you've just got to figure out how to calculate those sub-ranges to get efficient operation without exceeding the packet size limit.

MySQL++ already knows how to do that, too, with `Query::insertfrom()`. We gave it a different name instead of adding yet another `insert()` overload because it doesn't merely build the **INSERT** query, which you then `execute()`. It's more like `storein()`, in that it wraps the entire operation up in a single call. This feature is demonstrated in `examples/ssqls6.cpp`:

```
#include "cmdline.h"
#include "printdata.h"
#include "stock.h"

#include <fstream>

using namespace std;


// Breaks a given text line of tab-separated fields up into a list of
// strings.
static size_t
tokenize_line(const string& line, vector<mysqlpp::String>& strings)
{
    string field;
    strings.clear();

    istringstream iss(line);
    while (getline(iss, field, '\t')) {
        strings.push_back(mysqlpp::String(field));
    }

    return strings.size();
}


// Reads a tab-delimited text file, returning the data found therein
// as a vector of stock SSQLS objects.
static bool
read_stock_items(const char* filename, vector<stock>& stock_vector)
{
    ifstream input(filename);
    if (!input) {
        cerr << "Error opening input file '" << filename << "'" << endl;
        return false;
    }

    string line;
    vector<mysqlpp::String> strings;
    while (getline(input, line)) {
        if (tokenize_line(line, strings) == 6) {
            stock_vector.push_back(stock(string(strings[0]), strings[1],
                    strings[2], strings[3], strings[4], strings[5]));
        }
        else {
            cerr << "Error parsing input line (doesn't have 6 fields) " <<
                    "in file '" << filename << "'" << endl;
            cerr << "invalid line: '" << line << "'" << endl;
```

```
        }
    }

    return true;
}


int
main(int argc, char *argv[])
{
    // Get database access parameters from command line
    mysqlpp::examples::CommandLine cmdline(argc, argv);
    if (!cmdline) {
        return 1;
    }

    // Read in a tab-delimited file of stock data
    vector<stock> stock_vector;
    if (!read_stock_items("examples/stock.txt", stock_vector)) {
        return 1;
    }

    try {
        // Establish the connection to the database server.
        mysqlpp::Connection con(mysqlpp::examples::db_name,
                cmdline.server(), cmdline.user(), cmdline.pass());

        // Clear all existing rows from stock table, as we're about to
        // insert a bunch of new ones, and we want a clean slate.
        mysqlpp::Query query = con.query();
        query.exec("DELETE FROM stock");

        // Insert data read from the CSV file, allowing up to 1000
        // characters per packet.  We're using a small size in this
        // example just to force multiple inserts.  In a real program,
        // you'd want to use larger packets, for greater efficiency.
        mysqlpp::Query::MaxPacketInsertPolicy<> insert_policy(1000);
        query.insertfrom(stock_vector.begin(), stock_vector.end(),
                insert_policy);

        // Retrieve and print out the new table contents.
        print_stock_table(query);
    }
    catch (const mysqlpp::BadQuery& er) {
        // Handle any query errors
        cerr << "Query error: " << er.what() << endl;
        return -1;
    }
    catch (const mysqlpp::BadConversion& er) {
        // Handle bad conversions
        cerr << "Conversion error: " << er.what() << endl <<
                "\tretrieved data size: " << er.retrieved <<
                ", actual size: " << er.actual_size << endl;
        return -1;
    }
    catch (const mysqlpp::BadInsertPolicy& er) {
        // Handle bad conversions
        cerr << "InsertPolicy error: " << er.what() << endl;
        return -1;
    }
    catch (const mysqlpp::Exception& er) {
        // Catch-all for any other MySQL++ exceptions
        cerr << "Error: " << er.what() << endl;
```

```
        return -1;
    }

    return 0;
}
```

Most of the complexity in this example goes to just reading in the data from a file; we have to get our test data from somewhere. There are only two key lines of code: create an insertion policy object, and pass it along with an STL container full of row data to `Query::insertfrom()`.

This policy object is the main thing that differentiates `insertfrom()` from the two-iterator form of `insert()`. It controls how `insertfrom()` builds the query strings, primarily controlling how large each query gets before `insertfrom()` executes it and starts building a new query. We designed it to use policy objects because there is no single “right” choice for the decisions it makes.

MySQL++ ships with three different insertion policy classes, which should cover most situations.

`MaxPacketInsertPolicy`, demonstrated in the example above, does things the most obvious way: when you create it, you pass the maximum packet size, which it uses to prevent queries from going over the size limit. It builds up a query string row by row, checking each time through the loop whether adding another insert statement to the query string would make the packet size go over the limit. When that happens, or it gets to the end of the iteration range, it executes the query and starts over if it’s not yet at the end. This is robust, but it has a downside: it has to build each insert query in advance of knowing that it can append it to the larger query. Any time an insert query would push the packet over the limit, it has to throw it away, causing the library to do more work than is strictly necessary.

Imagine you’ve done some benchmarking and have found that the point of diminishing returns is at about 20 KB per query in your environment; beyond that point, the per-query overhead ceases to be an issue. Let’s also say you know for a fact that your largest row will always be less than 1 MB — less 20 KB — when expressed as a SQL insert statement. In that case, you can use the more efficient `SizeThresholdInsertPolicy`. It differs from `MaxPacketInsertPolicy` in that it allows `insertfrom()` to insert rows blindly into the query string until the built query exceeds the threshold, 20 KB in this example. Then it ships the packet off, and if successful, starts a new query. Thus, each query (except possibly the last) will be at least 20 KB, exceeding that only by as much as one row’s worth of data, minus one byte. This is quite appropriate behavior when your rows are relatively small, as is typical for tables not containing BLOB data. It is more efficient than `MaxPacketInsertPolicy` because it never has to throw away any SQL fragments.

The simplest policy object type is `RowCountInsertPolicy`. This lets you simply say how many rows at a time to insert into the database. This works well when you have a good handle on how big each row will be, so you can calculate in advance how many rows you can insert at once without exceeding some given limit. Say you know your rows can’t be any bigger than about 1 KB. If we stick with that 20 KB target, passing `RowCountInsertPolicy<>(20)` for the policy object would ensure we never exceed the size threshold. Or, say that maximum size value above is still true, but we also know the average row size is only 200 bytes. You could pass `RowCountInsertPolicy<>(100)` for the policy, knowing that the average packet size will be around 20 KB, and the worst case packet size 100 KB, still nowhere near the default 1 MB packet size limit. The code for this policy is very simple, so it makes your program a little smaller than if you used either of the above policies. Obviously it’s a bad choice if you aren’t able to predict the size of your rows accurately.

If one of the provided insert policy classes doesn’t suit your needs, you can easily create a custom one. Just study the implementation in `lib/insertpolicy.*`.

## Interaction with Transactions

These policy classes are all templates, taking a parameter that defaults to Transaction. This means that, by default, `insertfrom()` wraps the entire operation in a SQL transaction, so that if any of the insertions fail, the database

server rolls them all back. This prevents an error in the middle of the operation from leaving just part of the container's data inserted in the database, which you usually don't want any more than you'd want half a single row to be inserted.

There are good reasons why you might not want this. Perhaps the best reason is if the `insertfrom()` call is to be part of a larger transaction. MySQL doesn't support nested transactions, so the `insertfrom()` call will fail if it tries to start one of its own. You can pass `NoTransactions` for the insert policy's template parameter to make it suppress the transaction code.

# 5.5. Modifying data

It almost as easy to modify data with SSQLS as to add it. This is `examples/ssqls3.cpp`:

```
#include "cmdline.h"
#include "printdata.h"
#include "stock.h"

#include <iostream>

using namespace std;

int
main(int argc, char *argv[])
{
    // Get database access parameters from command line
    mysqlpp::examples::CommandLine cmdline(argc, argv);
    if (!cmdline) {
        return 1;
    }

    try {
        // Establish the connection to the database server.
        mysqlpp::Connection con(mysqlpp::examples::db_name,
                cmdline.server(), cmdline.user(), cmdline.pass());

        // Build a query to retrieve the stock item that has Unicode
        // characters encoded in UTF-8 form.
        mysqlpp::Query query = con.query("select * from stock ");
        query << "where item = " << mysqlpp::quote << "Nürnberger Brats";

        // Retrieve the row, throwing an exception if it fails.
        mysqlpp::StoreQueryResult res = query.store();
        if (res.empty()) {
            throw mysqlpp::BadQuery("UTF-8 bratwurst item not found in "
                    "table, run resetdb");
        }

        // Because there should only be one row in the result set,
        // there's no point in storing the result in an STL container.
        // We can store the first row directly into a stock structure
        // because one of an SSQLS's constructors takes a Row object.
        stock row = res[0];

        // Create a copy so that the replace query knows what the
        // original values are.
        stock orig_row = row;

        // Change the stock object's item to use only 7-bit ASCII, and
        // to deliberately be wider than normal column widths printed
        // by print_stock_table().
        row.item = "Nuerenberger Bratwurst";
```

```
        // Form the query to replace the row in the stock table.
        query.update(orig_row, row);

        // Show the query about to be executed.
        cout << "Query: " << query << endl;

        // Run the query with execute(), since UPDATE doesn't return a
        // result set.
        query.execute();

        // Retrieve and print out the new table contents.
        print_stock_table(query);
    }
    catch (const mysqlpp::BadQuery& er) {
        // Handle any query errors
        cerr << "Query error: " << er.what() << endl;
        return -1;
    }
    catch (const mysqlpp::BadConversion& er) {
        // Handle bad conversions
        cerr << "Conversion error: " << er.what() << endl <<
                "\tretrieved data size: " << er.retrieved <<
                ", actual size: " << er.actual_size << endl;
        return -1;
    }
    catch (const mysqlpp::Exception& er) {
        // Catch-all for any other MySQL++ exceptions
        cerr << "Error: " << er.what() << endl;
        return -1;
    }

    return 0;
}
```

Don't forget to run `resetdb` after running the example.

# 5.6. Storing SSQLSes in Associative Containers

One of the requirements of STL's associative containers on data stored in them is that the data type has to be less-than comparable. That is, it has to have an `operator <` defined. SSQLS does optionally give you this, as demonstrated in `examples/ssqls4.cpp`:

```
#include "cmdline.h"
#include "printdata.h"
#include "stock.h"

#include <iostream>

using namespace std;

int
main(int argc, char *argv[])
{
    // Get database access parameters from command line
    mysqlpp::examples::CommandLine cmdline(argc, argv);
    if (!cmdline) {
        return 1;
    }

    try {
        // Establish the connection to the database server.
        mysqlpp::Connection con(mysqlpp::examples::db_name,
```

```
                cmdline.server(), cmdline.user(), cmdline.pass());

        // Retrieve all rows from the stock table and put them in an
        // STL set.  Notice that this works just as well as storing them
        // in a vector, which we did in ssqls1.cpp.  It works because
        // SSQLS objects are less-than comparable.
        mysqlpp::Query query = con.query("select * from stock");
        set<stock> res;
        query.storein(res);

        // Display the result set.  Since it is an STL set and we set up
        // the SSQLS to compare based on the item column, the rows will
        // be sorted by item.
        print_stock_header(res.size());
        set<stock>::iterator it;
        cout.precision(3);
        for (it = res.begin(); it != res.end(); ++it) {
            print_stock_row(it->item.c_str(), it->num, it->weight,
                    it->price, it->sDate);
        }

        // Use set's find method to look up a stock item by item name.
        // This also uses the SSQLS comparison setup.
        it = res.find(stock("Hotdog Buns"));
        if (it != res.end()) {
            cout << endl << "Currently " << it->num <<
                    " hotdog buns in stock." << endl;
        }
        else {
            cout << endl << "Sorry, no hotdog buns in stock." << endl;
        }
    }
    catch (const mysqlpp::BadQuery& er) {
        // Handle any query errors
        cerr << "Query error: " << er.what() << endl;
        return -1;
    }
    catch (const mysqlpp::BadConversion& er) {
        // Handle bad conversions
        cerr << "Conversion error: " << er.what() << endl <<
                "\tretrieved data size: " << er.retrieved <<
                ", actual size: " << er.actual_size << endl;
        return -1;
    }
    catch (const mysqlpp::Exception& er) {
        // Catch-all for any other MySQL++ exceptions
        cerr << "Error: " << er.what() << endl;
        return -1;
    }

    return 0;
}
```

The `find()` call works because of the way the SSQLS was declared. It's properly covered elsewhere, but suffice it to say, the "1" in the declaration of `stock` above tells it that only the first field needs to be checked in comparing two SSQLSes. In database terms, this makes it the primary key. Therefore, when searching for a match, our exemplar only had to have its first field populated.

# 5.7. Changing the Table Name

Another feature you might find a use for is changing the table name MySQL++ uses to build queries involving SSQLSes. By default, the database server table is assumed to have the same name as the SSQLS structure type. But if this is inconvenient, you can globally change the table name used in queries like this:

```
stock::table("MyStockData");
```

It's also possible to change the name of a table on a per-instance basis:

```
stock s;
s.instance_table("AlternateTable");
```

This is useful when you have an SSQLS definition that is compatible with multiple tables, so the table name to use for each instance is different. This feature saves you from having to define a separate SSQLS for each table. It is also useful for mapping a class hierarchy onto a set of table definitions. The common SSQLS definition is the "superclass" for a given set of tables.

Strictly speaking, you only need to use this feature in multithreaded programs. Changing the static table name before using each instance is safe if all changes happen within a single thread. That said, it may still be convenient to change the name of the table for an SSQLS instance in a single-threaded program if it gets used for many operations over an extended span of code.

# 5.8. Using an SSQLS in Multiple Modules

It's convenient to define an SSQLS in a header file so you can use it in multiple modules. You run into a bit of a problem, though, because each SSQLS includes a few static data members to hold information common to all structures of that type. (The table name and the list of field names.) When you **#include** that header in more than one module, you get a multiply-defined symbol error at link time.

The way around this is to define the preprocessor macro `MYSQLPP_SSQLS_NO_STATICS` in *all but one* of the modules that use the header definining the SSQLS. When this macro is defined, it suppresses the static data members in any SSQLS defined thereafter.

Imagine we have a file `my_ssqls.h` which includes a `sql_create_N` macro call to define an SSQLS, and that that SSQLS is used in at least two modules. One we'll call `foo.cpp`, and we'll say it's just a user of the SSQLS; it doesn't "own" it. Another of the modules, `my_ssqls.cpp` uses the SSQLS more heavily, so we've called it the owner of the SSQLS. If there aren't very many modules, this works nicely:

```
// File foo.cpp, which just uses the SSQLS, but doesn't "own" it:
#define MYSQLPP_SSQLS_NO_STATICS
#include "my_ssqls.h"
```

```
// File my_ssqls.cpp, which owns the SSQLS, so we just #include it directly
#include "my_ssqls.h"
```

If there are many modules that need the SSQLS, adding all those **#defines** can be a pain. In that case, it's easier if you flip the above pattern on its head:

```
// File my_ssqls.h:
#if !defined(EXPAND_MY_SSQLS_STATICS)
#   define MYSQLPP_SSQLS_NO_STATICS
```

```
#endif
sql_create_X(Y, Z....) // the SSQLS definition


// File foo.cpp, a mere user of the SSQLS:
#include "my_ssqls.h"


// File my_ssqls.cpp, which owns the SSQLS:
#define EXPAND_MY_SSQLS_STATICS
#include "my_ssqls.h"
```

# 5.9. Harnessing SSQLS Internals

The sql_create macros define several methods for each SSQLS. These methods are mostly for use within the library, but some of them are useful enough that you might want to harness them for your own ends. Here is some pseudocode showing how the most useful of these methods would be defined for the stock structure used in all the `ssqls*.cpp` examples:

```
// Basic form
template <class Manip>
stock_value_list<Manip> value_list(cchar *d = ",",
  Manip m = mysqlpp::quote) const;

template <class Manip>
stock_field_list<Manip> field_list(cchar *d = ",",
  Manip m = mysqlpp::do_nothing) const;

template <class Manip>
stock_equal_list<Manip> equal_list(cchar *d = ",",
  cchar *e = " = ", Manip m = mysqlpp::quote) const;


// Boolean argument form
template <class Manip>
stock_cus_value_list<Manip> value_list([cchar *d, [Manip m,] ]
  bool i1, bool i2 = false, ... , bool i5 = false) const;

// List form
template <class Manip>
stock_cus_value_list<Manip> value_list([cchar *d, [Manip m,] ]
  stock_enum i1, stock_enum i2 = stock_NULL, ...,
  stock_enum i5 = stock_NULL) const;

// Vector form
template <class Manip>
stock_cus_value_list<Manip> value_list([cchar *d, [Manip m,] ]
  vector<bool> *i) const;

...Plus the obvious equivalents for field_list() and equal_list()
```

Rather than try to learn what all of these methods do at once, let's ease into the subject. Consider this code:

```
stock s("Dinner Rolls", 75, 0.95, 0.97, sql_date("1998-05-25"));
cout << "Value list: " << s.value_list() << endl;
cout << "Field list: " << s.field_list() << endl;
cout << "Equal list: " << s.equal_list() << endl;
```

That would produce something like:

```
Value list: 'Dinner Rolls',75,0.95,0.97,'1998-05-25'
Field list: item,num,weight,price,sdate
Equal list: item = 'Dinner Rolls',num = 75,weight = 0.95, price = 0.97,sdate = '1998-05-25'
```

That is, a "value list" is a list of data member values within a particular SSQLS instance, a "field list" is a list of the fields (columns) within that SSQLS, and an "equal list" is a list in the form of an SQL equals clause.

Just knowing that much, it shouldn't surprise you to learn that `Query::insert()` is implemented more or less like this:

```
*this << "INSERT INTO " << v.table() << " (" << v.field_list() <<
    ") VALUES (" << v.value_list() << ")";
```

where 'v' is the SSQLS you're asking the Query object to insert into the database.

Now let's look at a complete example, which uses one of the more complicated forms of `equal_list()`. This example builds a query with fewer hard-coded strings than the most obvious technique requires, which makes it more robust in the face of change. Here is `examples/ssqls5.cpp`:

```
#include "cmdline.h"
#include "printdata.h"
#include "stock.h"

#include <iostream>
#include <vector>

using namespace std;

int
main(int argc, char *argv[])
{
    // Get database access parameters from command line
    mysqlpp::examples::CommandLine cmdline(argc, argv);
    if (!cmdline) {
        return 1;
    }

    try {
        // Establish the connection to the database server.
        mysqlpp::Connection con(mysqlpp::examples::db_name,
                cmdline.server(), cmdline.user(), cmdline.pass());

        // Get all the rows in the stock table.
        mysqlpp::Query query = con.query("select * from stock");
        vector<stock> res;
        query.storein(res);

        if (res.size() > 0) {
            // Build a select query using the data from the first row
            // returned by our previous query.
            query << "select * from stock where " <<
                    res[0].equal_list(" and ", stock_weight, stock_price);

            // Display the finished query.
            cout << "Custom query:\n" << query << endl;
        }
    }
    catch (const mysqlpp::BadQuery& er) {
        // Handle any query errors
        cerr << "Query error: " << er.what() << endl;
```

```
        return -1;
    }
    catch (const mysqlpp::BadConversion& er) {
        // Handle bad conversions
        cerr << "Conversion error: " << er.what() << endl <<
                "\tretrieved data size: " << er.retrieved <<
                ", actual size: " << er.actual_size << endl;
        return -1;
    }
    catch (const mysqlpp::Exception& er) {
        // Catch-all for any other MySQL++ exceptions
        cerr << "Error: " << er.what() << endl;
        return -1;
    }

    return 0;
}
```

This example uses the list form of `equal_list()`. The arguments `stock_weight` and `stock_price` are enum values equal to the position of these columns within the stock table. sql_create_# generates this enum for you automatically.

The boolean argument form of that `equal_list()` call would look like this:

```
query << "select * from stock where " <<
    res[0].equal_list(" and ", false, false, true, true, false);
```

It's a little more verbose, as you can see. And if you want to get really complicated, use the vector form:

```
vector<bool> v(5, false);
v[stock_weight] = true;
v[stock_price] = true;
query << "select * from stock where " <<
    res[0].equal_list(" and ", v);
```

This form makes the most sense if you are building many other queries, and so can re-use that vector object.

Many of these methods accept manipulators and custom delimiters. The defaults are suitable for building SQL queries, but if you're using these methods in a different context, you may need to override these defaults. For instance, you could use these methods to dump data to a text file using different delimiters and quoting rules than SQL.

At this point, we've seen all the major aspects of the SSQLS feature. The final sections of this chapter look at some of the peripheral aspects.

# 5.10. Having Different Field Names in C++ and SQL

There's a more advanced SSQLS creation macro, which all the others are built on top of. Currently, the only feature it adds over what's described above is that it lets you name your SSQLS fields differently from the names used by the database server. Perhaps you want to use Hungarian notation in your C++ program without changing the SQL database schema:

```
sql_create_complete_5(stock, 1, 5,
    mysqlpp::sql_char, m_sItem, "item",
    mysqlpp::sql_bigint, m_nNum, "num",
    mysqlpp::sql_double, m_fWeight, "weight",
    mysqlpp::sql_decimal, m_fPrice, "price",
    mysqlpp::sql_date, m_Date, "sdate")
```

Note that you don't have to use this mechanism if the only difference in your SQL and C++ field names is case. SSQLS field name lookups are case-insensitive as of MySQL++ 3.1. You can see this in the examples: some parts of the code deliberately refer to the `stock.sdate` sample table field as `stock.sDate` to exercise this feature.

# 5.11. Expanding SSQLS Macros

If you ever need to see the code that a given SSQLS declaration expands out to, use the utility `doc/ssqls-pretty`, like so:

```
doc/ssqls-pretty < myprog.cpp | less
```

This Perl script locates the first SSQLS declaration in that file, then uses the C++ preprocessor to expand that macro. (The script assumes that your system's preprocessor is called `cpp`, and that its command line interface follows Unix conventions.)

If you run it from the top MySQL++ directory, as shown above, it will use the header files in the distribution's `lib` subdirectory. Otherwise, it assumes the MySQL++ headers are in their default location, `/usr/include/mysql++`. If you want to use headers in some other location, you'll need to change the directory name in the **-I** flag at the top of the script.

# 5.12. Customizing the SSQLS Mechanism

The SSQLS header `ssqls.h` is automatically generated by the Perl script `ssqls.pl`. Although it is possible to change this script to get additional functionality, most of the time it's better to just derive a custom class from the generated SSQLS to add functionality to it. (See the next section to see how to do this correctly.)

That said, `ssqls.pl` does have a few configurables you might want to tweak.

The first configurable value sets the maximum number of data members allowed in an SSQLS. This is discussed elsewhere, in Section 8.2, "The Maximum Number of Fields Allowed". Beware the warnings there about increasing this value too much.

The second configurable is the default floating point precision used for comparison. As described above (Section 5.2, "SSQLS Comparison and Initialization") SSQLSes can be compared for equality. The only place this is tricky is with floating-point numbers, since rounding errors can make two "equal" values compare as distinct. This property of floating-point numbers means we almost never want to do exact comparison. MySQL++ lets you specify the precision you want it to use. If the difference between two values is under a given threshold, MySQL++ considers the values equal. The default threshold is 0.00001. This threshold works well for "human" scale values, but because of the way floating-point numbers work, it can be wildly inappropriate for very large or very small quantities like those used in scientific applications.

There are actually two ways to change this threshold. If you need a different system-wide default, edit `ssqls.pl` and change the `$fp_min_delta` variable at the top of the file, then rebuild `ssqls.h` as described below. If you need different thresholds per file or per project, it's better to set the C macro `MYSQLPP_FP_MIN_DELTA` instead. The Perl variable sets this macro's default; if you give a different value before #including `ssqls.h`, it will use that instead.

To rebuild `ssqls.h` after changing `ssqls.pl`, you'll need a Perl interpreter. The only modern Unixy system I'm aware of where Perl isn't installed by default is Cygwin, and it's just a `setup.exe` choice away there. You'll probably only have to download and install a Perl interpreter if you're on Windows and don't want to use Cygwin.

If you're on a system that uses autoconf, building MySQL++ automatically updates `ssqls.h` any time `ssqls.pl` changes. Otherwise, you'll need to run the Perl interpreter by hand:

```
c:\mysql++> cd lib
c:\lib> perl ssqls.pl
```

# 5.13. Deriving from an SSQLS

Specialized SQL Structures make good base classes. They're simple, and have few requirements on any class that derives from them. There are some gotchas to look out for, however.

Consider this:

```
sql_create_2(
  Base, 1, 2,
  mysqlpp::sql_varchar, a,
  mysqlpp::sql_int, b
);

class Derived : public Base
{
public:
  // constructor
  Derived(mysqlpp::sql_varchar _a, mysqlpp::sql_int _b) :
  Base(_a, _b)
  {
  }

  // functionality added to the SSQLS through inheritance
  bool do_something_interesting(int data);
};
```

We've derived a class from an SSQLS in order to add a method to it. Easy, right?

Sadly, too easy. The code has a rather large flaw which makes our derived class unusable as an SSQLS. In C++, if a derived class has a function of the same name as one in the base class, the base class versions of that function are all hidden by those in the derived class. This applies to constructors, too: an SSQLS defines several constructors, but our derived class defines only one, causing that one to hide all of the ones in the base class. Many of the MySQL++ mechanisms that use SSQLSes rely on having these contructors, so our `Derived` above is-not-a `Base`, and so it isn't an SSQLS. If you try to use `Derived` as an SSQLS, you'll get compiler errors wherever MySQL++ tries to access one of these other constructors.

There's another minor flaw, as well. Our lone constructor above takes its parameters by value, but the corresponding constructor in the SSQLS takes them by const reference. Our derived class has technically hidden a fourth base class constructor this way, but this particular case is more a matter of efficiency than correctness. Code that needs the full-creation constructor will still work with our code above, but passing stringish types like `sql_varchar` by value instead of by const reference is inefficient.

This is the corrected version of the above code:

```
sql_create_2(
  Base, 1, 2,
  mysqlpp::sql_varchar, a,
  mysqlpp::sql_int, b
);

class Derived : public Base
{
public:
```

```
  // default constructor[14]
  Derived() :
  Base()
  {
  }

  // for-comparison constructor[15]
  Derived(const mysqlpp::sql_varchar& _a) :
  Base(_a)
  {
  }

  // full creation constructor
  Derived(const mysqlpp::sql_varchar& _a, const mysqlpp::sql_int& _b) :
  Base(_a, _b)
  {
  }

  // population constructor[16]
  Derived(const mysqlpp::Row& row) :
  Base(row)
  {
  }

  // functionality added to the SSQLS through inheritance
  bool do_something_interesting(int data);
};
```

Now `Derived` is-an SSQLS.

You might wonder if you can use protected inheritance above to redefine the SSQLS's public interface. For instance, OO purists might object to the public data members in an SSQLS. You could encapsulate these public data members in the derived class by using protected inheritance, exposing access to the base class's data members with public accessor methods. The problem with this is that each SSQLS has *dozens* of public member functions. These are needed by MySQL++ internals, so unless you re-exposed all of them as we did with the constructors above, you'd again have an SSQLS derivative that is-not-an SSQLS. Simply put, only public inheritance is practical with SSQLSes.

# 5.14. SSQLS and BLOB Columns

It takes special care to use SSQLS with BLOB columns. It's safest to declare the SSQLS field as of type `mysqlpp::sql_blob`. This is currently a typedef alias for String, which is the form the data is in just before the SSQLS mechanism populates the structure. Thus, when the data is copied from the internal MySQL++ data structures into your SSQLS, you get a direct copy of the `String` object's contents, without interference.

Because C++ strings handle binary data just fine, you might think you can use `std::string` instead of `sql_blob`, but the current design of `String` converts to `std::string` via a C string. As a result, the BLOB data is truncated at the first embedded null character during population of the SSQLS. There's no way to fix that without completely redesigning either `String` or the SSQLS mechanism.

The `sql_blob` typedef may be changed to alias a different type in the future, so using it instead of `String` ensures that your code tracks these library changes automatically. Besides, `String` is only intended to be an internal mechanism within MySQL++. The only reason the layering is so thin here is because it's the only way to prevent BLOB data from being corrupted while avoiding that looming redesign effort.

[14] needed by mechanisms like `Query::storein()`; anything using an STL container, which usually require default ctors for contained data structures

[15] takes the *`COMPCOUNT`* subset of the SSQLS's data members, used for making comparison exemplars, used with `Query::update()` and similar mechanisms; see Section 5.1, "sql_create" for more on *`COMPCOUNT`*

[16] used in taking raw row data from a SQL result set and converting it to SSQLS form

You can see this technique in action in the `cgi_jpeg` example:

```
#include "cmdline.h"
#include "images.h"

#define CRLF            "\r\n"
#define CRLF2           "\r\n\r\n"

int
main(int argc, char* argv[])
{
    // Get database access parameters from command line if present, else
    // use hard-coded values for true CGI case.
    mysqlpp::examples::CommandLine cmdline(argc, argv, "root",
            "nunyabinness");
    if (!cmdline) {
        return 1;
    }

    // Parse CGI query string environment variable to get image ID
    unsigned int img_id = 0;
    char* cgi_query = getenv("QUERY_STRING");
    if (cgi_query) {
        if ((strlen(cgi_query) < 4) || memcmp(cgi_query, "id=", 3)) {
            std::cout << "Content-type: text/plain" << std::endl << std::endl;
            std::cout << "ERROR: Bad query string" << std::endl;
            return 1;
        }
        else {
            img_id = atoi(cgi_query + 3);
        }
    }
    else {
        std::cerr << "Put this program into a web server's cgi-bin "
                "directory, then" << std::endl;
        std::cerr << "invoke it with a URL like this:" << std::endl;
        std::cerr << std::endl;
        std::cerr << "    http://server.name.com/cgi-bin/cgi_jpeg?id=2" <<
                std::endl;
        std::cerr << std::endl;
        std::cerr << "This will retrieve the image with ID 2." << std::endl;
        std::cerr << std::endl;
        std::cerr << "You will probably have to change some of the #defines "
                "at the top of" << std::endl;
        std::cerr << "examples/cgi_jpeg.cpp to allow the lookup to work." <<
                std::endl;
        return 1;
    }

    // Retrieve image from DB by ID
    try {
        mysqlpp::Connection con(mysqlpp::examples::db_name,
                cmdline.server(), cmdline.user(), cmdline.pass());
        mysqlpp::Query query = con.query();
        query << "SELECT * FROM images WHERE id = " << img_id;
        mysqlpp::StoreQueryResult res = query.store();
        if (res && res.num_rows()) {
            images img = res[0];
            if (img.data.is_null) {
                std::cout << "Content-type: text/plain" << CRLF2;
                std::cout << "No image content!" << CRLF;
            }
            else {
```

```
                std::cout << "X-Image-Id: " << img_id << CRLF; // for debugging
                std::cout << "Content-type: image/jpeg" << CRLF;
                std::cout << "Content-length: " <<
                        img.data.data.length() << CRLF2;
                std::cout << img.data;
            }
        }
        else {
            std::cout << "Content-type: text/plain" << CRLF2;
            std::cout << "ERROR: No image with ID " << img_id << CRLF;
        }
    }
    catch (const mysqlpp::BadQuery& er) {
        // Handle any query errors
        std::cout << "Content-type: text/plain" << CRLF2;
        std::cout << "QUERY ERROR: " << er.what() << CRLF;
        return 1;
    }
    catch (const mysqlpp::Exception& er) {
        // Catch-all for any other MySQL++ exceptions
        std::cout << "Content-type: text/plain" << CRLF2;
        std::cout << "GENERAL ERROR: " << er.what() << CRLF;
        return 1;
    }

    return 0;
}
```

# 5.15. SSQLS and Visual C++ 2003

SSQLS works on all platforms supported by MySQL++ except for Visual C++ 2003. (Because the rest of MySQL++ works just fine with Visual C++ 2003, we haven't removed this platform from the supported list entirely.)

If you do need SSQLS and are currently on Visual C++ 2003, you have these options:

1. The simplest option is to upgrade to a newer version of Visual C++. The compiler limitations that break SSQLS are all fixed in Visual C++ 2005 and newer. Visual C++ Express is free and is apparently here to stay; coupled with the free wxWidgets library, it lacks little compared to Visual C++ Professional. A bonus of using wxWidgets is that it's cross-platform and better-supported than MFC.

2. If you can't upgrade your compiler, you may be able to downgrade to MySQL++ v2.*x*. The SSQLS feature in these older versions worked with Visual C++ 2003, but didn't let you use a given SSQLS in more than one module in a program. If you can live with that limitation and have a Perl interpreter on your system, you can re-generate `lib/ssqls.h` to remove the multiple-module SSQLS support. To do this, you run the command **perl ssqls.pl -v** from within MySQL++'s `lib` subdirectory before you build and install the library.

3. There's a plan to replace the current SSQLS mechanism with an entirely new code base. Although this is being done primarily to get new features that are too difficult to add within the current design, it also means we'll have the chance to test step-by-step along the way that we don't reintroduce code that Visual C++ 2003 doesn't support. This may happen without you doing anything, but if there's someone on the team who cares about this, that will naturally increase the chances that it does happen.

# 6. Using Unicode with MySQL++

## 6.1. A Short History of Unicode

...with a focus on relevance to MySQL++

In the old days, computer operating systems only dealt with 8-bit character sets. That only allows for 256 possible characters, but the modern Western languages have more characters combined than that alone. Add in all the other languages of the world plus the various symbols people use in writing, and you have a real mess!

Since no standards body held sway over things like international character encoding in the early days of computing, many different character sets were invented. These character sets weren't even standardized between operating systems, so heaven help you if you needed to move localized Greek text on a DOS box to a Russian Macintosh! The only way we got any international communication done at all was to build standards on top of the common 7-bit ASCII subset. Either people used approximations like a plain "c" instead of the French "ç", or they invented things like HTML entities ("ç" in this case) to encode these additional characters using only 7-bit ASCII.

Unicode solves this problem. It encodes every character used for writing in the world, using up to 4 bytes per character. Before emoji became popular, the subset covering the most economically valuable cases fit into the lower 65536 code points, so you could encode most texts using only two bytes per character. Many nominally Unicode-aware programs only support this subset, called the Basic Multilingual Plane, or BMP.

Unfortunately, Unicode was invented about two decades too late for Unix and C. Those decades of legacy created an immense inertia preventing a widespread move away from 8-bit characters. MySQL and C++ come out of these older traditions, and so they share the same practical limitations. MySQL++ doesn't have any code in it for Unicode conversions, and it likely never will; it just passes data along unchanged from the underlying MySQL C API, so you still need to be aware of these underlying issues.

During the development of the Plan 9 operating system (a kind of successor to Unix) Ken Thompson invented the UTF-8 encoding. UTF-8 is a superset of 7-bit ASCII and is compatible with C strings, since it doesn't use 0 bytes anywhere as multi-byte Unicode encodings do. As a result, many programs that deal in text will cope with UTF-8 data even though they have no explicit support for UTF-8. Follow the last link above to see how the design of UTF-8 allows this.

## 6.2. Unicode in MySQL

Since MySQL comes out of the Unix world, and it predates the widespread use of UTF-8 in Unix, the early versinos of MySQL had no explicit support for Unicode. From the start, you could store raw UTF-8 strings, but it wouldn't know how to do things like sort a column of UTF-8 strings.

MySQL 4.1 added the first explicit support for Unicode. This version of MySQL supported only the BMP, meaning that if you told it to expect strings to be in UTF-8, it could only use up to 3 bytes per character.

MySQL 5.5 was the first release to completely support Unicode. Because the BMP-only Unicode support had been in the wild for about 6 years by that point, and changing to the new character set requires a table rebuild, the new one was called "utf8mb4" rather than change the longstanding meaning of "utf8" in MySQL. This release also added a new alias for the old UTF-8 subset character set, "utf8mb3."

Finally, in MySQL 8.0, "utf8mb4" became the default character set. For backwards compatibility, "utf8" remains an alias for "utf8mb3."

As of MySQL++ 3.2.4, we've defined the `MYSQLPP_UTF8_CS` and `MYSQLPP_UTF8_COL` macros which expand to "utf8mb4" and "utf8mb4_general_ci" when you build MySQL++ against MySQL 5.5 and newer and to "utf8" and

“utf8_general_ci” otherwise. We use these macros in our `resetdb` example; you're welcome to use them in your code as well.

# 6.3. Unicode on Unixy Systems

Linux and Unix have system-wide UTF-8 support these days. If your operating system is of 2001 or newer vintage, it probably has such support.

On such a system, the terminal I/O code understands UTF-8 encoded data, so your program doesn’t require any special code to correctly display a UTF-8 string. If you aren’t sure whether your system supports UTF-8 natively, just run the `simple1` example: if the first item has two high-ASCII characters in place of the “ü” in “Nürnberger Brats”, you know it’s not handling UTF-8.

If your Unix doesn’t support UTF-8 natively, it likely doesn’t support any form of Unicode at all, for the historical reasons I gave above. Therefore, you will have to convert the UTF-8 data to the local 8-bit character set. The standard Unix function `iconv()` can help here. If your system doesn’t have the `iconv()` facility, there is a free implementation available from the GNU Project. Another library you might check out is IBM’s ICU. This is rather heavy-weight, so if you just need basic conversions, `iconv()` should suffice.

# 6.4. Unicode on Windows

Each Windows API function that takes a string actually comes in two versions. One version supports only 1-byte “ANSI” characters (a superset of ASCII), so they end in 'A'. Windows also supports the 2-byte subset of Unicode called UCS-2[17]. Some call these “wide” characters, so the other set of functions end in 'W'. The `MessageBox()` API, for instance, is actually a macro, not a real function. If you define the UNICODE macro when building your program, the `MessageBox()` macro evaluates to `MessageBoxW()`; otherwise, to `MessageBoxA()`.

Since MySQL uses the UTF-8 Unicode encoding and Windows uses UCS-2, you must convert data when passing text between MySQL++ and the Windows API. Since there’s no point in trying for portability — no other OS I’m aware of uses UCS-2 — you might as well use platform-specific functions to do this translation. Since version 2.2.2, MySQL++ ships with two Visual C++ specific examples showing how to do this in a GUI program. (In earlier versions of MySQL++, we did Unicode conversion in the console mode programs, but this was unrealistic.)

How you handle Unicode data depends on whether you’re using the native Windows API, or the newer .NET API. First, the native case:

```
// Convert a C string in UTF-8 format to UCS-2 format.
void ToUCS2(LPTSTR pcOut, int nOutLen, const char* kpcIn)
{
  MultiByteToWideChar(CP_UTF8, 0, kpcIn, -1, pcOut, nOutLen);
}

// Convert a UCS-2 string to C string in UTF-8 format.
void ToUTF8(char* pcOut, int nOutLen, LPCWSTR kpcIn)
{
  WideCharToMultiByte(CP_UTF8, 0, kpcIn, -1, pcOut, nOutLen, 0, 0);
}
```

These functions leave out some important error checking, so see `examples/vstudio/mfc/mfc_dlg.cpp` for the complete version.

[17] Since Windows XP, Windows actually uses the UTF-16 encoding, not UCS-2. This means that if you use characters beyond the 16-bit BMP range, they get encoded as 4-byte characters. But again, since the most economically valuable subset of Unicode is the BMP if you ignore emoji, many programs ignore this distinction and assume Unicode strings on Windows are always 2 bytes per character.

If you're building a .NET application (such as, perhaps, because you're using Windows Forms), it's better to use the .NET libraries for this:

```
// Convert a C string in UTF-8 format to a .NET String in UCS-2 format.
String^ ToUCS2(const char* utf8)
{
  return gcnew String(utf8, 0, strlen(utf8), System::Text::Encoding::UTF8);
}

// Convert a .NET String in UCS-2 format to a C string in UTF-8 format.
System::Void ToUTF8(char* pcOut, int nOutLen, String^ sIn)
{
  array<Byte>^ bytes = System::Text::Encoding::UTF8->GetBytes(sIn);
  nOutLen = Math::Min(nOutLen - 1, bytes->Length);
  System::Runtime::InteropServices::Marshal::Copy(bytes, 0,
    IntPtr(pcOut), nOutLen);
  pcOut[nOutLen] = '\0';
}
```

Unlike the native API versions, these examples are complete, since the .NET platform handles a lot of things behind the scenes for us. We don't need any error-checking code for such simple routines.

All of this assumes you're using Windows NT or one of its direct descendants: Windows 2000, Windows XP, Windows Vista, Windows 7, or any "Server" variant of Windows. Windows 95 and its descendants (98, ME, and CE) do not support Unicode. They still have the 'W' APIs for compatibility, but they just smash the data down to 8-bit and call the 'A' version for you.

## 6.5. For More Information

The Unicode FAQs page has copious information on this complex topic.

When it comes to Unix and UTF-8 specific items, the UTF-8 and Unicode FAQ for Unix/Linux is a quicker way to find basic information.

# 7. Using MySQL++ in a Multithreaded Program

MySQL++ is not "thread safe" in any meaningful sense. MySQL++ contains very little code that actively prevents trouble with threads, and all of it is optional. We have done some work in MySQL++ to make thread safety *achievable*, but it doesn't come for free.

The main reason for this is that MySQL++ is generally I/O-bound, not processor-bound. That is, if your program's bottleneck is MySQL++, the ultimate cause is usually the I/O overhead of using a client-server database. Doubling the number of threads will just let your program get back to waiting for I/O twice as fast. Since threads are evil and generally can't help MySQL++, the only optional thread awareness features we turn on in the shipping version of MySQL++ are those few that have no practical negative consequences. Everything else is up to you, the programmer, to evaluate and enable as and when you need it.

We're going to assume that you are reading this chapter because you find yourself needing to use threads for some other reason than to speed up MySQL access. Our purpose here is limited to setting down the rules for avoiding problems with MySQL++ in a multi-threaded program. We won't go into the broader issues of thread safety outside the scope of MySQL++. You will need a grounding in threads in general to get the full value of this advice.

## 7.1. Build Issues

Before you can safely use MySQL++ with threads, there are several things you must do to get a thread-aware build:

1. *Build MySQL++ itself with thread awareness turned on.*

   On Linux, Cygwin and Unix (OS X, *BSD, Solaris...), pass the `--enable-thread-check` flag to the `configure` script. Beware, this is only a request to the `configure` script to look for thread support on your system, not a requirement to do or die: if the script doesn't find what it needs to do threading, MySQL++ will just get built without thread support. See `README-Unix.txt` for more details.

   On Windows, if you use the Visual C++ project files or the MinGW Makefile that comes with the MySQL++ distribution, threading is always turned on, due to the nature of Windows.

   If you build MySQL++ in some other way, such as with Dev-Cpp (based on MinGW) you're on your own to enable thread awareness.

2. *Link your program to a thread-aware build of the MySQL C API library.*

   If you use a binary distribution of MySQL on Unixy systems (including Cygwin) you usually get two different versions of the MySQL C API library, one with thread support and one without. These are typically called `libmysqlclient` and `libmysqlclient_r`, the latter being the thread-safe one. (The "`_r`" means reentrant.)

   If you're using the Windows binary distribution of MySQL, you should have only one version of the C API library, which should be thread-aware. If you have two, you probably just have separate debug and optimized builds. See `README-Visual-C++.txt` or `README-MinGW.txt` for details.

   If you build MySQL from source, you might only get one version of the MySQL C API library, and it can have thread awareness or not, depending on your configuration choices.

3. *Enable threading in your program's build options.*

   This is different for every platform, but it's usually the case that you don't get thread-aware builds by default. Depending on the platform, you might need to change compiler options, linker options, or both. See your development environment's documentation, or study how MySQL++ itself turns on thread-aware build options when requested.

# 7.2. Connection Management

The MySQL C API underpinning MySQL++ does not allow multiple concurrent queries on a single connection. You can run into this problem in a single-threaded program, too, which is why we cover the details elsewhere, in Section 3.16, "Concurrent Queries on a Connection". It's a thornier problem when using threads, though.

The simple fix is to just create a separarate Connection object for each thread that needs to make database queries. This works well if you have a small number of threads that need to make queries, and each thread uses its connection often enough that the server doesn't time out waiting for queries.

If you have lots of threads or the frequency of queries is low, the connection management overhead will be excessive. To avoid that, we created the ConnectionPool class. It manages a pool of `Connection` objects like library books: a thread checks one out, uses it, and then returns it to the pool as soon as it's done with it. This keeps the number of active connections low. We suggest that you keep each connection's use limited to a single variable scope for RAII reasons; we created a little helper called ScopedConnection to make that easy.

`ConnectionPool` has three methods that you need to override in a subclass to make it concrete: `create()`, `destroy()`, and `max_idle_time()`. These overrides let the base class delegate operations it can't successfully do itself to its subclass. The `ConnectionPool` can't know how to `create()` the `Connection` objects, because that depends on how your program gets login parameters, server information, etc. `ConnectionPool` also makes the subclass `destroy()` the `Connection` objects it created; it could assume that they're simply allocated on the heap with `new`, but it can't be sure, so the base class delegates destruction, too. Finally, the base class can't know which connection idle timeout policy would make the most sense to the client, so it asks its subclass via the `max_idle_time()` method.

`ConnectionPool` also allows you to override `release()`, if needed. For simple uses, it's not necessary to override this.

In designing your `ConnectionPool` derivative, you might consider making it a Singleton, since there should only be one pool in a program.

Another thing you might consider doing is passing a ReconnectOption object to `Connection::set_option()` in your `create()` override before returning the new `Connection` pointer. This will cause the underlying MySQL C API to try to reconnect to the database server if a query fails because the connection was dropped by the server. This can happen if the DB server is allowed to restart out from under your application. In many applications, this isn't allowed, or if it does happen, you might want your code to be able to detect it, so MySQL++ doesn't set this option for you automatically.

Here is an example showing how to use connection pools with threads:

```
#include "cmdline.h"
#include "threads.h"

#include <iostream>

using namespace std;


#if defined(HAVE_THREADS)
// Define a concrete ConnectionPool derivative.  Takes connection
// parameters as inputs to its ctor, which it uses to create the
// connections we're called upon to make.  Note that we also declare
// a global pointer to an object of this type, which we create soon
// after startup; this should be a common usage pattern, as what use
// are multiple pools?
class SimpleConnectionPool : public mysqlpp::ConnectionPool
{
```

```
public:
    // The object's only constructor
    SimpleConnectionPool(mysqlpp::examples::CommandLine& cl) :
    conns_in_use_(0),
    db_(mysqlpp::examples::db_name),
    server_(cl.server()),
    user_(cl.user()),
    password_(cl.pass())
    {
    }

    // The destructor.  We _must_ call ConnectionPool::clear() here,
    // because our superclass can't do it for us.
    ~SimpleConnectionPool()
    {
        clear();
    }

    // Do a simple form of in-use connection limiting: wait to return
    // a connection until there are a reasonably low number in use
    // already.  Can't do this in create() because we're interested in
    // connections actually in use, not those created.  Also note that
    // we keep our own count; ConnectionPool::size() isn't the same!
    mysqlpp::Connection* grab()
    {
        while (conns_in_use_ > 8) {
            cout.put('R'); cout.flush(); // indicate waiting for release
            sleep(1);
        }

        ++conns_in_use_;
        return mysqlpp::ConnectionPool::grab();
    }

    // Other half of in-use conn count limit
    void release(const mysqlpp::Connection* pc)
    {
        mysqlpp::ConnectionPool::release(pc);
        --conns_in_use_;
    }

protected:
    // Superclass overrides
    mysqlpp::Connection* create()
    {
        // Create connection using the parameters we were passed upon
        // creation.  This could be something much more complex, but for
        // the purposes of the example, this suffices.
        cout.put('C'); cout.flush(); // indicate connection creation
        return new mysqlpp::Connection(
                db_.empty() ? 0 : db_.c_str(),
                server_.empty() ? 0 : server_.c_str(),
                user_.empty() ? 0 : user_.c_str(),
                password_.empty() ? "" : password_.c_str());
    }

    void destroy(mysqlpp::Connection* cp)
    {
        // Our superclass can't know how we created the Connection, so
        // it delegates destruction to us, to be safe.
        cout.put('D'); cout.flush(); // indicate connection destruction
        delete cp;
    }
```

```
    unsigned int max_idle_time()
    {
        // Set our idle time at an example-friendly 3 seconds.  A real
        // pool would return some fraction of the server's connection
        // idle timeout instead.
        return 3;
    }

private:
    // Number of connections currently in use
    unsigned int conns_in_use_;

    // Our connection parameters
    std::string db_, server_, user_, password_;
};
SimpleConnectionPool* poolptr = 0;


static thread_return_t CALLBACK_SPECIFIER
worker_thread(thread_arg_t running_flag)
{
    // Ask the underlying C API to allocate any per-thread resources it
    // needs, in case it hasn't happened already.  In this particular
    // program, it's almost guaranteed that the safe_grab() call below
    // will create a new connection the first time through, and thus
    // allocate these resources implicitly, but there's a nonzero chance
    // that this won't happen.  Anyway, this is an example program,
    // meant to show good style, so we take the high road and ensure the
    // resources are allocated before we do any queries.
    mysqlpp::Connection::thread_start();
    cout.put('S'); cout.flush(); // indicate thread started

    // Pull data from the sample table a bunch of times, releasing the
    // connection we use each time.
    for (size_t i = 0; i < 6; ++i) {
        // Go get a free connection from the pool, or create a new one
        // if there are no free conns yet.  Uses safe_grab() to get a
        // connection from the pool that will be automatically returned
        // to the pool when this loop iteration finishes.
        mysqlpp::ScopedConnection cp(*poolptr, true);
        if (!cp) {
            cerr << "Failed to get a connection from the pool!" << endl;
            break;
        }

        // Pull a copy of the sample stock table and print a dot for
        // each row in the result set.
        mysqlpp::Query query(cp->query("select * from stock"));
        mysqlpp::StoreQueryResult res = query.store();
        for (size_t j = 0; j < res.num_rows(); ++j) {
            cout.put('.');
        }

        // Delay 1-4 seconds before doing it again.  Because this can
        // delay longer than the idle timeout, we'll occasionally force
        // the creation of a new connection on the next loop.
        sleep(rand() % 4 + 1);
    }

    // Tell main() that this thread is no longer running
    *reinterpret_cast<bool*>(running_flag) = false;
    cout.put('E'); cout.flush(); // indicate thread ended
```

```
    // Release the per-thread resources before we exit
    mysqlpp::Connection::thread_end();

    return 0;
}
#endif


int
main(int argc, char *argv[])
{
#if defined(HAVE_THREADS)
    // Get database access parameters from command line
    mysqlpp::examples::CommandLine cmdline(argc, argv);
    if (!cmdline) {
        return 1;
    }

    // Create the pool and grab a connection.  We do it partly to test
    // that the parameters are good before we start doing real work, and
    // partly because we need a Connection object to call thread_aware()
    // on to check that it's okay to start doing that real work.  This
    // latter check should never fail on Windows, but will fail on most
    // other systems unless you take positive steps to build with thread
    // awareness turned on.  See README-*.txt for your platform.
    poolptr = new SimpleConnectionPool(cmdline);
    try {
        mysqlpp::ScopedConnection cp(*poolptr, true);
        if (!cp->thread_aware()) {
            cerr << "MySQL++ wasn't built with thread awareness!  " <<
                    argv[0] << " can't run without it." << endl;
            return 1;
        }
    }
    catch (mysqlpp::Exception& e) {
        cerr << "Failed to set up initial pooled connection: " <<
                e.what() << endl;
        return 1;
    }

    // Setup complete.  Now let's spin some threads...
    cout << endl << "Pool created and working correctly.  Now to do "
            "some real work..." << endl;
    srand((unsigned int)time(0));
    bool running[] = {
            true, true, true, true, true, true, true,
            true, true, true, true, true, true, true };
    const size_t num_threads = sizeof(running) / sizeof(running[0]);
    size_t i;
    for (i = 0; i < num_threads; ++i) {
        if (int err = create_thread(worker_thread, running + i)) {
            cerr << "Failed to create thread " << i <<
                    ": error code " << err << endl;
            return 1;
        }
    }

    // Test the 'running' flags every second until we find that they're
    // all turned off, indicating that all threads are stopped.
    cout.put('W'); cout.flush(); // indicate waiting for completion
    do {
        sleep(1);
```

```
        i = 0;
        while (i < num_threads && !running[i]) ++i;
    }
    while (i < num_threads);
    cout << endl << "All threads stopped!" << endl;

    // Shut it all down...
    delete poolptr;
    cout << endl;
#else
    (void)argc;     // warning squisher
    cout << argv[0] << " requires that threads be enabled!" << endl;
#endif

    return 0;
}
```

The example works with both Windows native threads and with POSIX threads.[18] Because thread-enabled builds are only the default on Windows, it's quite possible for this program to do nothing on other platforms. See above for instructions on enabling a thread-aware build.

If you write your code without checks for thread support like you see in the code above and link it to a build of MySQL++ that isn't thread-aware, it will still try to run. The threading mechanisms fall back to a single-threaded mode when threads aren't available. A particular danger is that the mutex lock mechanism used to keep the pool's internal data consistent while multiple threads access it will just quietly become a no-op if MySQL++ is built without thread support. We do it this way because we don't want to make thread support a MySQL++ prerequisite. And, although it would be of limited value, this lets you use `ConnectionPool` in single-threaded programs.

You might wonder why we don't just work around this weakness in the C API transparently in MySQL++ instead of suggesting design guidelines to avoid it. We'd like to do just that, but how?

If you consider just the threaded case, you could argue for the use of mutexes to protect a connection from trying to execute two queries at once. The cure is worse than the disease: it turns a design error into a performance sap, as the second thread is blocked indefinitely waiting for the connection to free up. Much better to let the program get the "Commands out of sync" error, which will guide you to this section of the manual, which tells you how to avoid the error with a better design.

Another option would be to bury `ConnectionPool` functionality within MySQL++ itself, so the library could create new connections at need. That's no good because the above example is the most complex in MySQL++, so if it were mandatory to use connection pools, the whole library would be that much more complex to use. The whole point of MySQL++ is to make using the database easier. MySQL++ offers the connection pool mechanism for those that really need it, but an option it must remain.

## 7.3. Helper Functions

`Connection` has several thread-related static methods you might care about when using MySQL++ with threads.

You can call `Connection::thread_aware()` to determine whether MySQL++ and the underlying C API library were both built to be thread-aware. I want to stress that thread *awareness* is not the same thing as thread *safety*: it's still up to you to make your code thread-safe. If this method returns true, it just means it's *possible* to achieve thread-safety, not that you actually have it.

If your program's connection-management strategy allows a thread to use a `Connection` object that another thread created, you need to know about `Connection::thread_start()`. This function sets up per-thread resources needed to make MySQL server calls. You don't need to call it when you use the simple `Connection`-per-thread

[18] The file `examples/threads.h` contains a few macros and such to abstract away the differences between the two threading models.

strategy, because this function is implicitly called the first time you create a `Connection` in a thread. It's not harmful to call this function from a thread that previously created a `Connection`, just unnecessary. The only time it's necessary is when a thread can make calls to the database server on a `Connection` that another thread created and that thread hasn't already created a `Connection` itself.

If you use `ConnectionPool`, you should call `thread_start()` at the start of each worker thread because you probably can't reliably predict whether your `grab()` call will create a new `Connection` or will return one previously returned to the pool from another thread. It's possible to conceive of situations where you can guarantee that each pool user always creates a fresh `Connection` the first time it calls `grab()`, but thread programming is complex enough that it's best to take the safe path and always call `thread_start()` early in each worker thread.

Finally, there's the complementary method, `Connection::thread_end()`. Strictly speaking, it's not *necessary* to call this. The per-thread memory allocated by the C API is small, it doesn't grow over time, and a typical thread is going to need this memory for its entire run time. Memory debuggers aren't smart enough to know all this, though, so they will gripe about a memory leak unless you call this from each thread that uses MySQL++ before that thread exits.

Although its name suggests otherwise, `Connection::thread_id()` has nothing to do with anything in this chapter.

# 7.4. Sharing MySQL++ Data Structures

We're in the process of making it safer to share MySQL++'s data structures across threads. Although things are getting better, it's highly doubtful that all problems with this are now fixed. By way of illustration, allow me explain one aspect of this problem and how we solved it in MySQL++ 3.0.0.

When you issue a database query that returns rows, you also get information about the columns in each row. Since the column information is the same for each row in the result set, older versions of MySQL++ kept this information in the result set object, and each Row kept a pointer back to the result set object that created it so it could access this common data at need. This was fine as long as each result set object outlived the `Row` objects it returned. It required uncommon usage patterns to run into trouble in this area in a single-threaded program, but in a multi-threaded program it was easy. For example, there's frequently a desire to let one connection do the queries, and other threads process the results. You can see how avoiding lifetime problems here would require a careful locking strategy.

We got around this in MySQL++ v3.0 by giving these shared data structures a lifetime independent of the result set object that intitially creates it. These shared data structures stick around until the last object needing them gets destroyed.

Although this is now a solved problem, I bring it up because there are likely other similar lifetime and sequencing problems waiting to be discovered inside MySQL++. If you would like to help us find these, by all means, share data between threads willy-nilly. We welcome your crash reports on the MySQL++ mailing list. But if you'd prefer to avoid problems, it's better to keep all data about a query within a single thread. Between this and the advice in prior sections, you should be able to use threads with MySQL++ without trouble.

# 8. Configuring MySQL++

The default configuration of MySQL++ is suitable for most purposes, but there are a few things you can change to make it meet special needs.

## 8.1. The Location of the MySQL Development Files

MySQL++ is built on top of the MySQL C API. (Now called Connector/C.) MySQL++ relies on this low-level library for all communication with the database server. Consequently, the build process for MySQL++ may fail if it can't find the C API headers and library.

On platforms that use Autoconf[19], the `configure` script can usually figure out the location of the C API development files by itself[20] It simply tries a bunch of common installation locations until it finds one that works. If your MySQL server was installed in a nonstandard location, you will have to tell the `configure` script where these files are with some combination of the `--with-mysql`, `--with-mysql-include`, and `--with-mysql-lib` flags. See `README-Unix.txt` for details.

No other platform allows this sort of auto-discovery, so the build files for these platforms simply hard-code the default installation location for the current GA version of Connector/C at the time that version of MySQL++ was released. For example, the Visual C++ project files currently assume MySQL is in `c:\Program Files\MySQL\MySQL Server 5.1`. If you're using some other release of MySQL or you installed it somewhere else, you will have to modify the build files. How you do this, exactly, varies based on platform and what tools you have on hand. See `README-Visual-C++.txt`, `README-MinGW.txt`, or `README-Mac-OS-X.txt`, as appropriate.

## 8.2. The Maximum Number of Fields Allowed

MySQL++ offers two ways to automatically build SQL queries at run time: Template Queries and SSQLS. There's a limit on the number of fields these mechanisms support, defaulting to 25 fields in the official MySQL++ packages.[21] The files embodying these limits are `lib/querydef.h` and `lib/ssqls.h`, each generated by Perl scripts of the same name but with a `.pl` extension.

The default `querydef.h` is small and its size only increases linearly with respect to maximum field count.

`ssqls.h` is a totally different story. The default 25 field limit makes `ssqls.pl` generate an `ssqls.h` over 1 MB. Worse, the field limit to file size relation is *quadratic*.[22] This has a number of bad effects:

- Generating header files to support more fields than you actually require is a waste of space and bandwidth.
- Some compilers have arbitrary limits on the size of macros they're able to parse. Exceeding these limits usually causes the compiler to misbehave badly, rather than fail gracefully.
- Because it increases the size of two key files used in building MySQL++ itself and programs built on it, it increases compile times significantly. One test I did here showed a tripling of compile time from quadrupling the field limit.
- More than 25 fields in a table is a good sign of a bad database design, most likely a denormalization problem.

[19] Linux, Solaris, the BSDs, Mac OS X command line (as opposed to the Xcode IDE), Cygwin... Basically, Unix or anything that works like it.
[20] I don't say "Connector/C" here because the name change generally hasn't percolated out to Unixy systems. It's more commonly used on Windows systems, since the separate Connector/C download lets them avoid installing a MySQL server just to get development headers and libraries.
[21] If you're using a third-party MySQL++ package, its maintainer may have increased these field counts so the resulting headers more closely approach the size limit of the compiler the package was built with. In that case, you can look at the top of each generated header file to find out how many fields each supports.
[22] The file size equation, for you amateur mathematicians out there, is $N_{lines} = 18.5f^2 + 454.5f + 196.4$, where `f` is the field count.

The default limits try to mitigate against all of these factors while still being high enough to be useful with most DB designs.

If you're building MySQL++ from source on a platform that uses Autoconf, the easiest way to change these limits is at configuration time:

```
./configure --with-field-limit=50
```

That causes the configuration script to pass the **-f** flag to the two Perl scripts named above, overriding the default of 25 fields. Obviously you need a Perl interpreter on the system for this to work, but Perl is usually installed by default on systems MySQL++ supports via Autoconf.

On all other platforms, you'll have to give the **-f** flag to these scripts yourself. This may require installing Perl and putting it in the command path first. Having done that, you can do something like this to raise the limits:

```
cd lib
perl ssqls.pl -f 50
perl querydef.pl -f 50
```

Note the need to run these commands within the `lib` subdirectory of the MySQL++ source tree. (This is done for you automatically on systems where you are able to use the Autoconf method.)

# 8.3. Buried MySQL C API Headers

It's common these days on Unixy systems to install the MySQL C API headers in a `mysql` directory under some common `include` directory. If the C API headers are in `/usr/include/mysql`, we say they are "buried" underneath the system's main include directory, `/usr/include`. Since the MySQL++ headers depend on these C API headers, it can be useful for MySQL++ to know this fact.

When MySQL++ includes one of the C API headers, it normally does so in the obvious way:

```
#include <mysql.h>
```

But, if you define the `MYSQLPP_MYSQL_HEADERS_BURIED` macro, it switches to this style:

```
#include <mysql/mysql.h>
```

In common situations like the `/usr/include/mysql` one, this simplifies the include path options you pass to your compiler.

# 8.4. Building MySQL++ on Systems Without Complete C99 Support

MySQL++ uses the C99 header `stdint.h` for portable fixed-size integer typedefs where possible. The C99 extensions aren't yet officially part of the C++ Standard, so there are still some C++ compilers that don't offer this header. MySQL++ works around the lack of this header where it knows it needs to, but your platform might not be recognized, causing the build to break. If this happens, you can define the `MYSQLPP_NO_STDINT_H` macro to make MySQL++ use its best guess for suitable integer types instead of relying on `stdint.h`.

MySQL++ also uses C99's long long data type where available. MySQL++ has workarounds for platforms where this is known not to be available, but if you get errors in `common.h` about this type, you can define the macro `MYSQLPP_NO_LONG_LONGS` to make MySQL++ fall back to portable constructs.

# 9. Using MySQL++ in Your Own Project

Up to now, this manual has only discussed MySQL++ in conjunction with the example programs that come with the library. This chapter covers the steps you need to take to incorporate MySQL++ into your own projects.

The first thing you have to do is include `mysql++.h` in each module that uses MySQL++. In modules that use SSQLS v1, you also need to include `ssqls.h`.[23]

At this point, your project probably still won't compile, and it certainly won't link. The remaining steps are dependent on the operating system and tools you are using. The rest of this chapter is broken up into several sections, one for each major platform type. You can skip over the sections for platforms you don't use.

## 9.1. Visual C++

### Using MySQL++ in an MFC Project

If you don't already have a project set up, open Visual Studio, say File | New | Project, then choose Visual C++ | MFC | MFC Application. Go through the wizard setting up the project as you see fit.

Once you have your project open, right click on your top-level executable in the Solution Explorer, choose Properties, and make the following changes. (Where it doesn't specify Debug or Release, make the same change to both configurations.)

- Append the following to C/C++ | General | Additional Include Directories: `C:\Program Files\MySQL\MySQL Connector C 6.1\include, C:\mysql++\include`
- Under C/C++ | Code Generation change "Runtime Library" to "Multi-threaded Debug DLL (/MDd)" for the Debug configuration. For the Release configuration, make it "Multi-threaded DLL (/MD)".
- For both Release and Debug builds, append the following to Linker | General | Additional Library Directories: `C:\Program Files\MySQL\MySQL Connector C 6.1\lib, C:\mysql++\lib`

  Connector/C does include debug libraries, but you will probably not need to use them.
- Under Linker | Input add the following to "Additional Dependencies" for the Debug configuration: `libmysql.lib wsock32.lib mysqlpp_d.lib`

  ...and then for the Release configuration: `libmysql.lib wsock32.lib mysqlpp.lib`

  This difference is because MySQL++'s Debug DLL and import library have a `_d` suffix so you can have both in the same directory without conflicts.

You may want to study `examples\vstudio\mfc\mfc.vcproj` to see this in action. Note that some of the paths will be different, because it can use relative paths for `mysqlpp.dll`.

### Using MySQL++ in a Windows Forms C++/CLI Project

Before you start work on getting MySQL++ working with your own program, you need to make some changes to the MySQL++ build settings. Open `mysqlpp.sln`, then right-click on the mysqlpp target and select Properties. Make the following changes for both the Debug and Release configurations:

[23] MySQL++ has many header files, but the only one that isn't intertwined with the rest is `ssqls.h`. `mysql++.h` brings in all of the others in the correct order. Some have tried to speed their build times by finding a subset of MySQL++ headers to include, but `mysql++.h` already does as much of this as is practical. MySQL++'s monolithic nature rules out finding a true subset of the library headers.

- Under Configuration Properties | General, change “Common Language Runtime support” to the /clr setting.
- Under C/C++ | Code Generation, change “Enable C++ Exceptions” from “Yes (/EHsc)” to “Yes With SEH Exceptions (/EHa)”

If you have already built MySQL++, be sure to perform a complete rebuild after changing these options. The compiler will emit several C4835 warnings after making those changes, which are harmless when using the DLL with a C++/CLI program, but which warn of real problems when using it with unmanaged C++. This is why MySQL++’s Windows installer (`install.hta`) offers the option to install the CLR version into a separate directory; use it if you need both managed and unmanaged versions installed!

For the same reason, you might give some thought about where you install `mysqlpp.dll` on your end user’s machines when distributing your program. My recommendation is to install it in the same directory as the `.exe` file that uses it, rather than installing into a system directory where it could conflict with a `mysqlpp.dll` built with different settings.

Once you have MySQL++ built with CLR support, open your program’s project. If you don’t already have a project set up, open Visual Studio, say File | New | Project, then choose Visual C++ | CLR | Windows Forms Application. Go through the wizard setting up the project as you see fit.

The configuration process isn’t much different from that for an MFC project, so go through the list above first. Then, make the following changes particular to .NET and C++/CLI:

- Under Configuration Properties | General change the setting from /clr:pure to /clr. (You need mixed assembly support to allow a C++/CLI program to use a plain C++ library like MySQL++.)
- For the Linker | Input settings, you don’t need `wsock32.lib`. The mere fact that you’re using .NET takes care of that dependency for you.

In the MFC instructions above, it said that you need to build it using the Multi-threaded DLL version of the C++ Runtime Library. That’s not strictly true for MFC, but it’s an absolute requirement for C++/CLI. See the Remarks in the MSDN article on the /clr switch for details.

You may want to study `examples\vstudio\wforms\wforms.vcproj` to see all this in action. Note that some of the paths will be different, because it can use relative paths for `mysqlpp_d.dll` and `mysqlpp.dll`.

# 9.2. Unixy Platforms: Linux, *BSD, OS X, Cygwin, Solaris...

There are lots of ways to build programs on Unixy platforms. We’ll cover just the most generic way here, `Makefile`s. We’ll use a very simple example so it’s clear how to translate this to more sophisticated build systems such as GNU Autotools or Bakefile.

“Hello, world!” for MySQL++ might look something like this:

```
#include <mysql++.h>

int main()
{
    mysqlpp::String greeting("Hello, world!");
    std::cout << greeting << std::endl;
    return 0;
}
```

Here’s a `Makefile` for building that program:

```
CXXFLAGS := -I/usr/include/mysql -I/usr/local/include/mysql++
LDFLAGS := -L/usr/local/lib
```

```
LDLIBS := -lmysqlpp -lmysqlclient
EXECUTABLE := hello

all: $(EXECUTABLE)

clean:
    rm -f $(EXECUTABLE) *.o
```

The `*FLAGS` lines are where all of the assumptions about file and path names are laid out. Probably at least one of these assumptions isn't true for your system, and so will require changing.

The trickiest line is the `LDLIBS` one. MySQL++ programs need to get built against both the MySQL and MySQL++ libraries, because MySQL++ is built on top of the MySQL C API library[24] If you're building a threaded program, use `-lmysqlclient_r` instead of `-lmysqlclient` here. (See Section 7, "Using MySQL++ in a Multithreaded Program" for more details on building thread-aware programs.)

On some systems, the order of libraries in the `LDLIBS` line is important: these linkers collect symbols from right to left, so the rightmost library needs to be the most generic. In this example, MySQL++ depends on MySQL, so the MySQL C API library is rightmost.

You might need to add more libraries to the `LDLIBS` line. `-lnsl`, `-lz` and `-lm` are common. If you study how MySQL++ itself gets built on your system, you can see what it uses, and emulate that.

You may be wondering why we have used both `LDLIBS` and `LDFLAGS` here. Some `Makefiles` you have seen collect both types of flags in a single variable. That can work if the variable is used in the right place in the link command. However, this particular `Makefile` is made with GNU make in mind, and uses its standard rules implicitly. Those rules are designed to use these two variables separately like this. If you were writing your own compilation rules, you could write them in such a way that you didn't have to do this.

Beyond that, we have a pretty vanilla `Makefile`, thanks in large part to the fact that the default `make` rules are fine for such a simple program.

# 9.3. OS X

## Makefiles

The generic `Makefile` instructions above cover most of what you need to know about using Makefiles on OS X.

One thing that may trip you up on OS X is that it uses an uncommon dynamic linkage system. The easiest way to cope with this is to link your executables with the compiler, rather than call `ld` directly.

Another tricky bit on OS X is the concept of Universal binaries. See `README-Mac-OS-X.txt` for details on building a Universal version of the MySQL++ library, if you need one. By default, you only get a version tuned for the system type you build it on.

## Xcode

I have no information on how to incorporate MySQL++ in an Xcode project. Send a message to the MySQL++ mailing list if you can help out here.

[24]The MySQL C API library is most commonly called `libmysqlclient` on Unixy systems, though it is also known as Connector/C.

# 9.4. MinGW

## Makefiles

The generic `Makefile` instructions above apply to MinGW's version of GNU `make` as well. You will have some differences due to the platform, so here's the adjusted `Makefile`:

```
SHELL := $(COMSPEC)
MYSQL_DIR := "c:/Program Files/MySQL/MySQL Connector C 6.1"
CXXFLAGS := -I$(MYSQL_DIR)/include -Ic:/MySQL++/include
LDFLAGS := -L$(MYSQL_DIR)/lib -Lc:/MySQL++/lib/MinGW
LDLIBS := -lmysql -lmysqlpp
EXECUTABLE := hello

all: $(EXECUTABLE)

clean:
 del $(EXECUTABLE)
```

Note that I've used **del** instead of **rm** in the clean target. In the past, at least, MinGW `make` had some funny rules about whether commands in target rules would get run with `sh.exe` or with `cmd.exe`. I can't currently get my installation of MinGW to do anything but use `sh.exe` by default, but that may be because I have Cygwin installed, which provides `sh.exe`. This explains the first line in the file, which overrides the default shell with `cmd.exe`, purely to get consistent behavior across platforms. If you knew all your platforms would have a better shell, you'd probably want to use that instead.

Note the use of forward slashes in the path to the MySQL Connector/C development files. GNU `make` uses the backslash as an escape character, so you'd have to double them if you're unwilling to use forward slashes.

## Third-Party MinGW IDEs (Dev-C++, Code::Blocks...)

I have no information on how to do this. We've received reports on the mailing list from people that have made it work, but no specifics on what all needs to be done. The `Makefile` discussion above should give you some hints.

# 9.5. Eclipse

As far as I can tell, the simplest way to build a C++ project with Eclipse is to set up a `Makefile` for it as described above, then add an external run configuration for your local `make` tool. Get the project building from the command line with `make`, then go to Run | External Tools | Open External Tools Dialog and add a new launch configuration.

For example, on my OS X system I use `/usr/bin/gnumake` for the program location and pick the project root with the Browse Workspace button to set the working directory.

# 10. Incompatible Library Changes

This chapter documents those library changes since the epochal 1.7.9 release that break end-user programs. You can dig this stuff out of the `ChangeLog.md` file, but the change log focuses more on explaining and justifying the facets of each change, while this section focuses on how to migrate your code between these library versions.

Since pure additions do not break programs, those changes are still documented only in the change log.

## 10.1. API Changes

This section documents files, functions, methods and classes that were removed or changed in an incompatible way. If your program uses the changed item, you will have to change something in your program to get it to compile after upgrading to each of these versions.

### v1.7.10

Removed `Row::operator[]()` overloads except the one for size_type, and added `Row::lookup_by_name()` to provide the "subscript by string" functionality. In practical terms, this change means that the `row["field"]` syntax no longer works; you must use the new `lookup_by_name` method instead.

Renamed the generated library on POSIX systems from `libsqlplus` to `libmysqlpp`.

### v1.7.19

Removed `SQLQuery::operator=()`, and the same for its `Query` subclass. Use the copy constructor instead, if you need to copy one query to another query object.

### v1.7.20

The library used to have two names for many core classes: a short one, such as `Row` and a longer one, `MysqlRow`. The library now uses the shorter names exclusively.

All symbols within MySQL++ are in the `mysqlpp` namespace now if you use the new `mysql++.h` header. If you use the older `sqlplus.hh` or `mysql++.hh` headers, these symbols are hoist up into the global namespace. The older headers cause the compiler to emit warnings if you use them, and they will go away someday.

### v2.0.0

#### Connection class changes

- `Connection::create_db()` and `drop_db()` return true on success. They returned false in v1.7.*x*! This change will only affect your code if you have exceptions disabled.
- Renamed `Connection::real_connect()` to `connect()`, made several more of its parameters default, and removed the old `connect()` method, as it's now a strict subset of the new one. The only practical consequence is that if your program was using `real_connect()`, you will have to change it to `connect()`.
- Replaced `Connection::read_option()` with new `set_option()` mechanism. In addition to changing the name, programs using this function will have to use the new `Connection::Option` enumerated values, accept a true return value as meaning success instead of 0, and use the proper argument type. Regarding the latter, `read_option()` took a const char* argument, but because it was just a thin wrapper over the MySQL C API function mysql-options, the actual value being pointed to could be any of several types. This new mechanism is properly type-safe.

## Exception-related changes

- Classes `Connection`, `Query`, `Result`, `ResUse`, and `Row` now derive from OptionalExceptions which gives these classes a common interface for disabling exceptions. In addition, almost all of the per-method exception-disabling flags were removed. The preferred method for disabling exceptions on these objects is to create an instance of the new NoExceptions class on the stack, which disables exceptions on an `OptionalExceptions` subclass as long as the `NoExceptions` instance is in scope. You can instead call `disable_exceptions()` on any of these objects, but if you only want them disabled temporarily, it's easy to forget to re-enable them later.

- In the previous version of MySQL++, those classes that supported optional exceptions that could create instances of other such classes were supposed to pass this flag on to their children. That is, if you created a `Connection` object with exceptions enabled, and then asked it to create a `Query` object, the `Query` object also had exceptions disabled. The problem is, this didn't happen in all cases where it should have in v1.7. This bug is fixed in v2.0. If your program begins crashing due to uncaught exceptions after upgrading to v2.0, this is the most likely cause. The most expeditious fix in this situation is to use the new `NoExceptions` feature to return these code paths to the v1.7 behavior. A better fix is to rework your program to avoid or deal with the new exceptions.

- All custom MySQL++ exceptions now derive from the new Exception interface. The practical upshot of this is that the variability between the various exception types has been eliminated. For instance, to get the error string, the `BadQuery` exception had a string member called `error` plus a method called `what()`. Both did the same thing, and the `what()` method is more common, so the error string was dropped from the interface. None of the example programs had to be changed to work with the new exceptions, so if your program handles MySQL++ exceptions the same way they do, your program won't need to change, either.

- Renamed `SQLQueryNEParams` exception to `BadParamCount` to match style of other exception names.

- Added BadOption, ConnectionFailed, DBSelectionFailed, EndOfResults, EndOfResultSets, LockFailed, and ObjectNotInitialized exception types, to fix overuse of `BadQuery`. Now the latter is used only for errors on query execution. If your program has a "catch-all" block taking a `std::exception` for each try block containing MySQL++ statements, you probably won't need to change your program. Otherwise, the new exceptions will likely show up as program crashes due to unhandled exceptions.

## Query class changes

- In previous versions, `Connection` had a querying interface similar to class `Query`'s. These methods were intended only for `Query`'s use; no example ever used this interface directly, so no end-user code is likely to be affected by this change.

- A more likely problem arising from the above change is code that tests for query success by calling the `Connection` object's `success()` method or by casting it to bool. This will now give misleading results, because queries no longer go through the `Connection` object. Class `Query` has the same success-testing interface, so use it instead.

- `Query` now derives from `std::ostream` instead of `std::stringstream`.

## Result/ResUse class changes

- Renamed `ResUse::mysql_result()` to `raw_result()` so it's database server neutral.

- Removed `ResUse::eof()`, as it wrapped the deprecated and unnecessary MySQL C API function mysql-eof. See the `simple3` and `usequery` examples to see the proper way to test for the end of a result set.

## Row class changes

- Removed "field name" form of `Row::field_list()`. It was pointless.

- `Row` subscripting works more like v1.7.9: one can subscript a `Row` with a string (e.g. `row["myfield"]`), or with an integer (e.g. `row[5]`). `lookup_by_name()` was removed. Because `row[0]` is ambiguous (0 could mean the first field, or be a null pointer to const char*), there is now `Row::at()`, which can look up any field by index.

### Miscellaneous changes

- Where possible, all distributed Makefiles only build dynamic libraries. (Shared objects on most Unices, DLLs on Windows, etc.) Unless your program is licensed under the GPL or LGPL, you shouldn't have been using the static libraries from previous versions anyway.
- Removed the backwards-compatibility headers `sqlplus.hh` and `mysql++.hh`. If you were still using these, you will have to change to `mysql++.h`, which will put all symbols in namespace mysqlpp.
- Can no longer use arrow operator (->) on the iterators into the `Fields`, `Result` and `Row` containers.

## v2.2.0

Code like this will have to change:

```
query << "delete from mytable where myfield=%0:myvalue";
query.parse();
query.def["myvalue"] = some_value;
query.execute();
```

...to something more like this:

```
query << "delete from mytable where myfield=%0";
query.parse();
query.execute(some_value);
```

The first code snippet abuses the default template query parameter mechanism (`Query::def`) to fill out the template instead of using one of the overloaded forms of `execute()`, `store()` or `use()` taking one or more `SQLString` parameters. The purpose of `Query::def` is to allow for default template parameters over multiple queries. In the first snippet above, there is only one parameter, so in order to justify the use of template queries in the first place, it must be changing with each query. Therefore, it isn't really a "default" parameter at all. We did not make this change maliciously, but you can understand why we are not in any hurry to restore this "feature".

(Incidentally, this change was made to allow better support for BLOB columns.)

## v2.3.0

`Connection::set_option()` calls now set the connection option immediately, instead of waiting until just before the connnection is actually established. Code that relied on the old behavior could see unhandled exceptions, since option setting errors are now thrown from a different part of the code. You want to wrap the actual `set_option()` call now, not `Connection::connect()`

`FieldNames` and `FieldTypes` are no longer exported from the library. If you are using these classes directly from Visual C++ or MinGW, your code won't be able to dynamically link to a DLL version of the library any more. These are internal classes, however, so no one should be using them directly.

# v3.0.0

## Class name changes

Several classes changed names in this release:

- `ColData` is now `String`.
- `NullisBlank` is now `NullIsBlank`. (Note the capital *I*.) Similar changes for `NullisNull` and `NullisZero`.
- `ResNSel` is now `SimpleResult`.
- `Result` is now `StoreQueryResult`.
- `ResUse` is now `UseQueryResult`.
- `SQLString` is now `SQLTypeAdapter`.

When first building existing code against this version, you may find it helpful to define the macro `MYSQLPP_OLD_CLASS_NAMES` in your program's build options. This will turn on some macros that set up aliases for the new class names matching their corresponding old names. Then, when you've fixed up any other issues that may prevent your program from building with the new MySQL++, you can turn it back off and fix up any class name differences.

If you were only using `ColData` in a BLOB context, you should use `sql_blob` or one of the related typedefs defined in `lib/sql_types.h` instead, to insulate your code from changes like these.

The `SQLString` change shouldn't affect you, as this class was not designed to be used by end user code. But, due to the old name and the fact that it used to derive from `std::string`, some might have been tempted to use it as an enhanced `std::string`. Such code will undoubtedly break, but can probably be fixed by just changing it to use `std::string` instead.

## Connection class changes

The option setting mechanism has been redesigned. (Yes, again.) There used to be an enum in `Connection` with a value for each option we understood, and an overload of `Connection::set_option()` for each argument type we understood. It was possible to pass any option value to any `set_option()` overload, and the problem would only be detected at run time. Now each option is represented by a class derived from the new `Option` abstract base class, and `set_option()` simply takes a pointer to one of these objects. See `examples/multiquery.cpp` for the syntax. Since each `Option` subclass takes only the parameter types it actually understands, it's now completely type-safe at compile time.

The new option setting mechanism also has the virtue of being more powerful so it let us replace several existing things within `Connection` with new options:

- Replaced `enable_ssl()` with `SslOption`.
- Replaced the `compress` parameter to the `Connection` create-and-connect constructor and `Connection::connect()` method with `CompressOption`.
- Replaced the `connect_timeout` parameter with `ConnectTimeoutOption`.
- Defined `Option` subclasses for each of the flags you would previously set using the `client_flag` parameter. There are about a dozen of these, so instead of listing them, look in `lib/options.h` for something with a similar name.

Collapsed `Connection`'s `host`, `port`, and `socket_name` parameters down into a new combined `server` parameter which is parsed to determine what kind of connection you mean. These interfaces are still compatible with v2.3 and earlier up through the port parameter.

Moved `Connection::affected_rows()`, `info()` and `insert_id()` methods to class `Query`, as they relate to the most recently-executed query.

Changed the return type of `Connection::ping()` from int to bool. If you were calling `ping()` in bool context or using its return value in bool context, you will need to reverse the sense of the test because the previous return code used zero to mean success. Now it returns true to indicate success.

Renamed several methods:

- Use `client_version()` instead of `api_version()` or `client_info()`.
- Use `ipc_version()` instead of `host_info()`.
- Use `protocol_version()` instead of `proto_info()`.
- Use `server_version()` instead of `server_info()`.
- Use `status()` instead of `stat()`.

Also, removed `close()` in favor of `disconnect()`, which has always done the same thing.

## Date and Time class changes

The sql_timestamp typedef is now an alias for `DateTime`, not `Time`.

There used to be implicit conversion constructors from `ColData` (now `String`), `std::string` and const char* for the `Date`, `DateTime`, and `Time` classes. It's still possible to do these conversions, but only explicitly. (This had to be done to make `Null<T>` work in SSQLSes.)

The most likely place to run into problems as a result of this change is in code like this:

```
void some_function(const mysqlpp::DateTime& dt);

some_function("2007-12-22");
```

The function call needs to be changed to:

```
some_function(mysqlpp::DateTime("2007-12-22"));
```

## Exception changes

If an error occurs during the processing of a "use" query (as opposed to the initial execution) we throw the new `UseQueryError` exception instead of `BadQuery`.

If you pass bad values to the `Row` ctor so that it can't initialize itself properly, it throws the `ObjectNotInitialized` exception instead of `BadQuery`.

Together, these two changes mean that `BadQuery` is now used solely to indicate a problem executing the actual SQL query statement.

## Field and Fields class changes

`Field` is now a real C++ class, not just a typedef for the corresponding C API class. Major portability impacts are:

- It has no public data members. Where sensible, there is a public accessor function of the same name as the corresponding field in the C API structure.
- The main exception to this is the `flags` data member. This is a bitfield in the C API data structure and you had to use MySQL-specific constants to break values out of it. MySQL++'s new `Field` class provides a public member function returning bool for each of these flags.
- The new class doesn't include all of the data members from the C API version. We left out those that aren't used within MySQL++ or its examples, or whose function we couldn't understand. Basically, if we couldn't document a reason to use it, we left it out.

`Fields` used to be a `std::vector` work-alike which worked with the C API to access fields and return them as though they were simply contained directly within the `Fields` object. Now that we have a real MySQL++ class to hold information about each field without reference to the C API, we were able to replace the `Fields` class with:

```
typedef std::vector<Field> Fields;
```

If anything, this should give a pure superset of the old functionality, but it's possible it could break end user code.

## Query class changes

If you were using char as an 8-bit integer in query building, there are several places in MySQL++ v3 where it will now be treated as a single-character string. MySQL++ has had the `tiny_int` class for many years now specifically to provide a true 8-bit integer without the semantic confusion surrounding the old C char type. Either use `tiny_int`, or use the SQL type aliases sql_tinyint and sql_tinyint_unsigned instead.

The 'r' and 'R' template query parameter modifiers were removed. They made the library do quoting and both quoting and escaping (respectively) regardless of the data type of the parameter. There are no corresponding `Query` stream manipulators, so for symmetery we had to decide whether to add such manipulators or remove the tquery modifiers. There should never be a reason to force quoting or escaping other than to work around a MySQL++ bug, and it's better to just fix the bug than work around it, so removed the tquery modifiers.

`Query::store_next()` and `Result::fetch_row()` no longer throw the `EndOfResults` and `EndOfResultSets` exceptions; these are not exceptional conditions! These methods simply return false when you hit the end of the result set now.

Renamed `Query::def` to `Query::template_defaults` to make its purpose clearer.

Removed `Query::preview()`. The most direct replacement for this set of overloaded methods is the parallel set of `str()` methods, which were just aliases before. (Chose `str()` over `preview()` because it's standard C++ nomenclature.) But if you're just looking to get a copy of a built query string and you aren't using template queries, you can now insert the `Query` into a stream and get the same result.

For example, a lot of code in the examples that used to say things like:

```
cout << query.preview() << endl;
```

now looks like this:

```
cout << query << endl;
```

## Result, ResUse, and ResNSel class changes

In addition to the class name changes described above, `UseQueryResult` is no longer `StoreQueryResult`'s base class. There is a new abstract class called `ResultBase` containing much of what used to be in `ResUse`, and it is the base of both of these concrete result set types. This should only affect your code if you were using `ResUse` references to refer to `Result` objects.

Removed a bunch of duplicate methods:

- Use `num_fields()` instead of `columns()`.
- Use `field_names()` instead of `names()`.
- Use `num_rows()` instead of `rows()`.
- Use `field_types()` instead of `types()`.

Renamed several methods for "grammar" reasons. For example, some methods returned a single object but had a "plural" name, implying that it returned a container of objects. In cases like this, we changed the name to agree with the return value. Some of these also fall into the duplicate method category above:

- Use `field(unsigned int)` instead of `fields(unsigned int)`.
- Use `field_num(const std::string&)` instead of `names(const std::string&)`.
- Use `field_name(int)` instead of `names(int)`.
- Use `field_type(int)` instead of `types(int)`.

Removed several "smelly" methods:

- `purge()`: was an internal implementation detail, not something for end user code to call
- `raw_result()`: end user code shouldn't be digging down to the C API data structures, but if you really need something like this, look at the implementation of `Query::storein()`. Its workings will probably be educational.
- `reset_names()`: no reason to call this, especially now that the field name list is initialized once at startup and then never changed
- `reset_field_names()`: just an alias for previous
- `reset_types()`: same argument as for `reset_names()`
- `reset_field_types()`: just an alias for previous

`ResUse::field_num()` would unconditionally throw a `BadFieldName` exception when you asked for a field that doesn't exist. Now, if exceptions are disabled on the object, it just returns -1.

`SimpleResult`'s member variables are all now private, and have read-only accessor functions of the same name.

Code like this used to work:

```
mysqlpp::Row row;
mysqlpp::Result::size_type i;
for (i = 0; row = res[i]; ++i) {
  // Do something with row here
}
```

That is, indexing past the end of a "store" result set would just return an empty row object, which tests as false in bool context, so it ends the loop. Now that `StoreQueryResult` is a `std::vector` derivative, this either crashes your program or causes the standard library to throw an exception, depending on what debugging features your version of STL has. The proper technique is:

```
mysqlpp::Row row;
mysqlpp::StoreQueryResult::size_type i;
for (i = 0; i < res.num_rows(); ++i) {
  row = res[i];
  // Do something with row here
}
```

...or, in a more C++ish idiom:

```
mysqlpp::Row row;
mysqlpp::StoreQueryResult::const_iterator it;
for (it = res.begin(); it != res.end(); ++it) {
  row = *it;
  // Do something with row here
}
```

## Row class changes

Removed `Row::raw_data()`, `raw_size()` and `raw_string()`. These were useful with BLOB data back when MySQL++ didn't handle embedded null characters very well, and when copies of `ColData` objects were expensive. Neither is true now, so they have no value any more. Equivalent calls are:

```
mysqlpp::String s = row[0];
s.data();                           // raw_data() equivalent
s.length();                         // raw_size() equivalent
std::string(s.data(), s.length()); // raw_string() equivalent
```

`Row::operator[](const char*)` would unconditionally throw a `BadFieldName` exception when you asked for a field that doesn't exist. Now, if exceptions are disabled on the `Row` object, it just returns a reference to an empty `String` object. You can tell when this happens because such an object tests as false in bool context.

## Specialized SQL Structure (SSQLS) changes

Renamed `custom*` to `ssqls*`. There is a backwards-compatibility header `custom.h` which includes `ssqls.h` for you, but it will go away in a future version of MySQL++.

SSQLSes get populated by field name now, not by field order. In v2, it was absolutely required that your SSQLS had its fields declared in exactly the same order as the fields in the database server, and there could be no gaps. An **ALTER TABLE** command would almost always necessitate redefining the corresponding SSQLS and rebuilding your program. Some alterations actually made using SSQLS impossible. For the most part, this change just gives your program additional flexibility in the face of future changes. However, code that was taking advantage of this low-level fact will break when moving to v3. Before I explain how, let's go over the high-level functional changes you'll find in v3's SSQLS mechanism.

Because MySQL++ no longer needs the SSQLS field order to match the SQL field order, the `sql_create_c_order_*` SSQLS creation macro was dropped in v3. We were also able to drop the ordering parameters from `sql_create_complete_*`. That in turn means there is no longer a difference between the way it and `sql_create_c_names_*` work, so the latter was also dropped. Thus, there are now only two groups of SSQLS creation macros left: `sql_create_*`, which works pretty much as it always has, and `sql_create_complete_*`, which is the same except for the lack of ordering parameters.

In general, you should be using `sql_create_*` for all SSQLSes unless you need to use different names for data members in C++ than you use for the corresponding columns in SQL. In that case, use `sql_create_complete_*` instead.

In v2, it was possible to have different SQL column names than SSQLS data member names while still using `sql_create_*` if you only used SSQLS for data retrieval.[25] In v3, you must use `sql_create_complete_*` for absolutely all uses of SSQLS when you want the C++ field names to differ from the SQL column names.

The new `Null<T>` support in SSQLSes causes an internal compiler error in Visual C++ 2003. (VC++ 2005 and newer have no trobule with it.) A poll on the mailing list says there aren't many people still stuck on this version, so we just ifdef'd out the SSQLS mechanism and all the examples that use it when built with VC++ 2003. If this affects you, see Section 5.15, "SSQLS and Visual C++ 2003" for suggestions on ways to cope.

If you are using types other than MySQL++'s sql_* ones [26] in your SSQLSes, code that previously worked may now see `TypeLookupFailed` exceptions. (This can be thrown even if exceptions are otherwise disabled in MySQL++.) This version of MySQL++ is stricter about mapping SQL to C++ type information, and vice versa. If the library can't find a suitable mapping from one type system to the other, it throws this exception, because its only other option would be to crash or raise an assertion. This typically happens when building SQL queries, so you can probably handle it the same way as if the subsequent query excecution failed. If you're catching the generic `mysqlpp::Exception`, your error handling code might not need to change. If you see this exception, it does mean you need to look into your use of data types, though. The table that controls this is `mysql_type_info::types`, defined at the top of `lib/type_info.cpp`. Every data type in `lib/sql_types.h` has a corresponding record in this table, so if you stick to those types, you'll be fine. It's also okay to use types your C++ compiler can convert directly to these predefined types.

The `_table` static member variable for each SSQLS is now private. The recommended way to access this remains unchanged: the `table()` static member function.

`table()` used to return a modifiable reference to the table name. Now there are two overloads, one which returns an unmodifiable pointer to the table name, and the other which takes const char* so you can override the default table name. So, the code we used to recommend for changing the SSQLS's table name:

```
my_ssqls_type::table() = "MyTableName";
```

now needs to be:

```
my_ssqls_type::table("MyTableName");
```

## Miscellaneous changes

MySQL++ does quoting and escaping much more selectively now. Basically, if the library can tell you're not building a SQL query using one of the standard methods, it assumes you're outputting values for human consumption, so it disables quoting and SQL escaping. If you need to build your own mechanism to replace this, quoting is easy to do, and `Query::escape_string()` can do SQL escaping for you.

Removed `success()` in `Connection`, `Query` and `SimpleResult` (neé `ResNSel`) and simply made these classes testable in bool context to get the same information. An additional change in `Connection` is that it used to

[25] In MySQL++ v2, data retreival (`Query::storein()`, `SSQLS(const Row& other)`, etc.) worked fine regardless of whether your SSQLS field names matched those in the corresponding SQL table, because the SSQLS was populated by position, not by field name. Thus, if all you used SSQLS for was data retrieval, you could define your structures with `sql_create_*` in v2. This was never recommended, because such an SSQLS wouldn't work with other features of MySQL++ like `Query::insert()` because they depend on being able to map names from C++ to SQL and back. You needed to use `sql_create_c_names_*` to make these features work in v2 in the face of a naming scheme difference between C++ and SQL.

[26] These typedefs have been available since MySQL++ v2.1.

be considered “unsuccessful” when the connection was down. Since the sense of this test is now whether the object is in a good state, it only returns false when the connection attempt fails. Call `Connection::is_connected()` if you just want to test whether the connection is up.

The debug mode build of the library now has a "_d" suffix for Visual C++, and Xcode. This lets you have both versions installed without conflict. The release build uses the current naming scheme. If you have an existing program building against MySQL++ on these platforms, you’ll need to change your build options to use the new name in debug mode.

Renamed `NO_LONG_LONGS` to `MYSQLPP_NO_LONG_LONGS` to avoid a risk of collision in the global macro namespace.

### v3.0.7

Most MySQL++ classes with `at()` or `operator []()` methods now throw the new BadIndex exception when you pass an out-of-range index. These methods variously either did not check their indices, or threw `std::out_of_range` when passed a bad index.

I say “most” because there is at least one MySQL++ class that doesn’t follow this rule. `Fields` is just a typedef for a specialization of `std::vector`, and the Standard has its own rules for index checking.

## 10.2. ABI Changes

This section documents those library changes that require you to rebuild your program so that it will link with the new library. Most of the items in the previous section are also ABI changes, but this section is only for those items that shouldn’t require any code changes in your program.

If you were going to rebuild your program after installing the new library anyway, you can probably ignore this section.

### v1.7.18

The `Query` classes now subclass from `stringstream` instead of the deprecated `strstream`.

### v1.7.19

Fixed several const-incorrectnesses in the `Query` classes.

### v1.7.22

Removed “reset query” parameters from several `Query` class members. This is not an API change, because the parameters were given default values, and the library would ignore any value other than the default. So, any program that tried to make them take another value wouldn’t have worked anyway.

### v1.7.24

Some freestanding functions didn’t get moved into namespace mysqlpp when that namespace was created. This release fixed that. It doesn’t affect the API if your program’s C++ source files say using namespace mysqlpp within them.

### v2.0.0

Removed `Connection::infoo()`. (I’d call this an API change if I thought there were any programs out there actually using this...)

Collapsed the `Connection` constructor taking a bool (for setting the throw_exceptions flag) and the default constructor into a single constructor using a default for the parameter.

Classes `Connection` and `Query` are now derived from the `Lockable` interface, instead of implementing their own lock/unlock functions.

In several instances, functions that took objects by value now take them by const reference, for efficiency.

Merged `SQLQuery` class's members into class `Query`.

Merged `RowTemplate` class's members into class `Row`.

Reordered member variable declarations in some classes. The most common instance is when the private section was declared before the public section; it is now the opposite way. This can change the object's layout in memory, so a program linking to the library must be rebuilt.

Simplified the date and time class hierarchy. Date used to derive from `mysql_date`, Time used to derive from `mysql_time`, and DateTime used to derive from both of those. All three of these classes used to derive from `mysql_dt_base`. All of the `mysql_*` classes' functionality and data has been folded into the leaf classes, and now the only thing shared between them is their dependence on the DTbase template. Since the leaf classes' interface has not changed and end-user code shouldn't have been using the other classes, this shouldn't affect the API in any practical way.

`mysql_type_info` now always initializes its private `num` member. Previously, this would go uninitialized if you used the default constructor. Now there is no default ctor, but the ctor taking one argument (which sets `num`) has a default.

## v3.0.0

Removed `reset_query` parameters from `Query` member functions. None of these have been honored at least going back to v1.7.9, so this is not an API change. As of this version, `Query` now automatically detects when it can safely reset itself after executing a query, so it's not necessary to ask for a reset except when using template queries.

Removed overloads of `Query::execute()`, `store()`, and `use()` that take only a const char*. This is not an API change because there was an equivalent call chain for this already. This change just snaps a layer of indirection.

`Query::error()` is now const and returns const char* instead of a `std::string` by value.

Removed `Lockable` mechanism as it was conceptually flawed. `Connection` and `Query` consequently no longer derive from `Lockable`. Since it was basically useless in prior versions, it can't be construed as an API change.

## v3.0.1

`Connection::thread_aware()`, `thread_start()` and `thread_end()` are now static methods, so a program can call them before creating a connection. Ditto for `DBDriver` methods of the same name.

`ConnectionPool::release()` is now virtual, so a subclass can override it.

## v3.0.2

`ConnectionPool::grab()` is now virtual; same reason as above.

`Query` can now be tested in bool context, as was intended for v3.0.0. Had to change the "safe bool" method signature to make it happen, so technically it's an API change, but it's still used the same way.

## v3.1.0

The addition of a few new virtual methods to `ConnectionPool` inadvertently changed the library ABI. I knew adding fields changed the ABI, but erroneously assumed that the inverse of that truth — that adding *methods* was always safe — was also true. Adding normal methods *is* safe, but adding *virtual* methods breaks the ABI because it changes the class's vtable size.

That left us with two bad choices: either we could come out with a 3.1.1 that removed these methods to restore the prior ABI, or we could just declare this the "new ABI" and move on, resolving not to fall into this trap again. We've chosen the latter path.

# 11. Licensing

The primary copyright holders on the MySQL++ library and its documentation are Kevin Atkinson (1998), MySQL AB (1999 through 2001) and Educational Technology Resources, Inc. (2004 through the date of this writing). There are other contributors, who also retain copyrights on their additions; see the `ChangeLog.md` file in the MySQL++ distribution tarball for details.

The MySQL++ library and its Reference Manual are released under the GNU Lesser General Public License (LGPL), reproduced below.

The MySQL++ User Manual — excepting some example code from the library reproduced within it — is offered under a license closely based on the Linux Documentation Project License (LDPL) v2.0, included below. (The MySQL++ documentation isn't actually part of the Linux Documentation Project, so the main changes are to LDP-related language. Also, generic language such as "author's (or authors')" has been replaced with specific language, because the license applies to only this one document.)

These licenses basically state that you are free to use, distribute and modify these works, whether for personal or commercial purposes, as long as you grant the same rights to those you distribute the works to, whether you changed them or not. See the licenses below for full details.

# 11.1. GNU Lesser General Public License

Version 2.1, February 1999



[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

## Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages--typically libraries--of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author’s reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the “Lesser” General Public License because it does Less to protect the user’s freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users’ freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a “work based on the library” and a “work that uses the library”. The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

# GNU LESSER GENERAL PUBLIC LICENSE

## TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called “this License”). Each licensee is addressed as “you”.

A “library” means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The “Library”, below, refers to any such software library or work which has been distributed under these terms. A “work based on the Library” means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term “modification”.)

“Source code” for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library’s complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

> a) The modified work must itself be a software library.
>
> b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
>
> c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
>
> d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.
>
> (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a “work that uses the Library”. Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a “work that uses the Library” with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a “work that uses the library”. The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a “work that uses the Library” uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a “work that uses the Library” with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer’s own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

> a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable “work that uses the Library”, as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)
>
> b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user’s computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.
>
> c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
>
> d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.
>
> e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the “work that uses the Library” must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

> a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.
>
> b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients’ exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

## How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the library's name and a brief idea of what it does.>

> Copyright © <year> <name of author>
>
> This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.
>
> This library 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 Lesser General Public License for more details.
>
> You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the library, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the library `Frob' (a library for tweaking knobs) written by James Random Hacker.
>
> <signature of Ty Coon>, 1 April 1990
>
> Ty Coon, President of Vice

That’s all there is to it!

# 11.2. MySQL++ User Manual License

## I. COPYRIGHT

The copyright to the MySQL++ User Manual is owned by its authors.

## II. LICENSE

The MySQL++ User Manual may be reproduced and distributed in whole or in part, in any medium physical or electronic, provided that this license notice is displayed in the reproduction. Commercial redistribution is permitted and encouraged. Thirty days advance notice via email to the authors of redistribution is appreciated, to give the authors time to provide updated documents.

### A. REQUIREMENTS OF MODIFIED WORKS

All modified documents, including translations, anthologies, and partial documents, must meet the following requirements:

1. The modified version must be labeled as such.
2. The person making the modifications must be identified.
3. Acknowledgement of the original author must be retained.
4. The location of the original unmodified document be identified.
5. The original authors' names may not be used to assert or imply endorsement of the resulting document without the original authors' permission.

In addition it is requested that:

1. The modifications (including deletions) be noted.
2. The authors be notified by email of the modification in advance of redistribution, if an email address is provided in the document.

Mere aggregation of the MySQL++ User Manual with other documents or programs on the same media shall not cause this license to apply to those other works.

All translations, derivative documents, or modified documents that incorporate the MySQL++ User Manual may not have more restrictive license terms than these, except that you may require distributors to make the resulting document available in source format.