MySQL++ User Manual

Kevin Atkinson
Sinisa Milivojevic
Monty Widenius
Warren Young

$Date: 2007-06-28 00:17:10 -0600 (Thu, 28 Jun 2007) $

Table of Contents

1. Introduction

MySQL++ is a powerful C++ wrapper for MySQL's C API. 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.

Then in 1999, Sinisa Milivojevic unofficially took over maintenance of the library, releasing versions 1.0 and 1.1. (All of Kevin's releases were pre-1.0 point releases.) 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. Since Sinisa is an employee of MySQL AB, it seems to be during this time that the dream of multiple-database compatibility died.

With version 1.7.9, MySQL++ went into a period of stasis, lasting over three years. During this time, Sinisa ran the MySQL++ mailing list and supported its users, but made no new releases. There were many patches submitted during this period, some of which were ignored, others which were just put on the MySQL++ web site for people to try. A lot of these patches were mutually-incompatible, and not all of them gave a fully-functional copy of MySQL++.

In early August of 2004, the current maintainer (Warren Young) got fed up with this situation and took over. He released 1.7.10 later that month.

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 developed into a very complex and powerful library, with many different ways to accomplish the same task. Unfortunately, this means that figuring out how to perform a simple task can be frustrating for new users. In this section we will provide an overview of the most important user-facing components of the library.

The overall process for using MySQL++ is similar to that of most other database access APIs:

1. Open the connection
2. Form and execute the query
3. Iterate through the result set
4. Go to 2 :)

There is, however, a lot of extra functionality along each step of the way.

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.

2.2. The Query Object

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

`Query` is subclassed from `std::stringstream` which means you can write to it like any other C++ stream to form a query. The library includes stream manipulators that make it easy to generate syntactically-correct SQL.

You can also set up Template Queries with this class. Template queries 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 Specialized SQL Structures (SSQLS). This feature presents your results as a C++ data structure, instead of making you access the data through MySQL++ intermediary classes. It also reduces the amount of embedded SQL code your program needs.

2.3. Result Sets

The field data in a result set are stored in a special `std::string`-like class called ColData. 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 (ResNSel) 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: Dynamic Method

The easiest way to retrieve data from MySQL uses a Result object, which includes one or more Row objects. Because these classes are `std::vector`-like containers, you can treat the result set as a two-dimensional array. For example, you can get the 5th item on the 2nd row by simply saying `result.at(1).at(4)`. You can also access row elements by field name, like this: `result.at(2)["price"]`.

An alternate way of accessing your query results is through a ResUse object. This class acts more like an STL input iterator than a container: you walk through your result set one item 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. This method is more efficient when there can be arbitrarily many results, which could pose a memory allocation problem with the previous technique.

Queries That Return Data: Static Method

The Specialized SQL Structures (SSQLS) feature method above defines C++ structures that match the table structures in your database schema.

We call it the "static" method because the table structure is fixed at compile time. Indeed, some schema changes require that you update your SSQLS definitions and recompile, or else the program could crash or throw "bad conversion" exceptions when MySQL++ tries to stuff the new data into an outdated data structure. (Not all changes require a recompile. Adding a column to a table is safe, for instance, as the program will ignore the new column until you update the SSQLS definition.)

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 a sequential or set-associative STL container for you. Consider this:

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

Isn't that slick?

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

This tutorial is meant to give you a jump start in using MySQL++. While it is a very complicated and powerful library, it's possible to make quite functional programs without tapping but a fraction of its power. This section will introduce you to the most useful fraction.

This tutorial assumes you know C++ fairly well, in particular the Standard Template Library (STL) and exceptions.

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 installed it via the RPM package, the example source code and a simplified Makefile is in the examples subdirectory of the mysql++-devel package's documentation directory. (This is usually `/usr/share/doc/mysql++-devel-*`, but it can vary on different Linuxes.)

Before you get started, please read through any of the README* 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 run it like so:

```
./exrun resetdb [host [user [password [port]]]]
```

`exrun` is a shell script that ensures that the MySQL++ example program you give as its first argument finds the correct shared library version. If you run the example program directly, it will search the system directories for the MySQL++ shared library. That will only work correctly if you've installed the library before running the examples. You should run the examples before installing the library to ensure that the library is working correctly, thus `exrun`. See `README.examples` for more details. (We've been using POSIX file and path names for simplicity above, but there's a Windows version of `exrun`, called `exrun.bat`. It works the same way.)

As for the remaining program arguments, they are all optional, but they must be in the order listed. If you leave off the port number, it uses the default value, 3306. If you leave off the password, it assumes you don't need one to log in. If you leave off the user name, it uses the name you used when logging on to the computer. And if you leave off the host name, it assumes the MySQL server is running on the local host. A typical invocation is:

```
./exrun resetdb localhost root nunyabinness
```

For `resetdb`, the user name needs to be for an account with permission to create databases. Once the database is created, you can use any account that has read and write permissions for the sample database, `mysql_cpp_data`.

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

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 "util.h"

#include <mysql++.h>

#include <iostream>
#include <iomanip>

using namespace std;

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

```
// Connect to the sample database.
mysqlpp::Connection con(false);
if (!connect_to_db(argc, argv, con)) {
return 1;
}

// Retrieve a subset of the sample stock table set up by resetdb
mysqlpp::Query query = con.query();
query << "select item from stock";
mysqlpp::Result res = query.store();

// Display the result set
cout << "We have:" << endl;
if (res) {
mysqlpp::Row row;
mysqlpp::Row::size_type i;
for (i = 0; row = res.at(i); ++i) {
cout << '\t' << row.at(0) << endl;
}
}
else {
cerr << "Failed to get item list: " << query.error() << endl;
return 1;
}

return 0;
}
```

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

Notice that MySQL++'s Result objects work similarly to the STL `std::vector` container. The only trick is that you can't use subscripting notation if the argument is ever 0, because of the way we do overloading, so it's safer to call `at()` instead.

The only thing that isn't explicit in the code above is that we delegate command line argument parsing and connection establishment to `connect_to_db()` in the `util` module. This function exists to give the examples a consistent interface, not to hide important details.

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 "util.h"

#include <mysql++.h>

#include <iostream>
#include <iomanip>

using namespace std;

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

```
// Connect to the sample database.
mysqlpp::Connection con(false);
if (!connect_to_db(argc, argv, con)) {
return 1;
}

// Retrieve the sample stock table set up by resetdb
mysqlpp::Query query = con.query();
query << "select * from stock";
mysqlpp::Result res = query.store();

// Display results
if (res) {
// Display header
cout.setf(ios::left);
cout << setw(21) << "Item" <<
setw(10) << "Num" <<
setw(10) << "Weight" <<
setw(10) << "Price" <<
"Date" << endl << endl;

// Get each row in result set, and print its contents
mysqlpp::Row row;
mysqlpp::Row::size_type i;
for (i = 0; row = res.at(i); ++i) {
cout << setw(20) << row["item"] << ' ' <<
setw(9) << row["num"] << ' ' <<
setw(9) << row["weight"] << ' ' <<
setw(9) << row["price"] << ' ' <<
setw(9) << row["sdate"] <<
endl;
}
}
else {
cerr << "Failed to get stock table: " << query.error() << endl;
return 1;
}

return 0;
}
```

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. Most of the examples have a full set of exception handlers. This is worthy of emulation.

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` by reference. 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.

Some of these exceptions are optional. When exceptions are disabled on a MySQL++ object, it signals errors in some other way, typically by returning an error code or setting an error flag. Classes that support this feature derive from OptionalExceptions. Moreover, when such an object 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. You can see this technique at work in the `simple[1-3]` examples, which keeps them, well, simple.

Real-world code typically can't afford to lose out on the additional information and control offered by exceptions. But at the same time, it is still sometimes useful to disable exceptions temporarily. 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. See `examples/resetdb.cpp` to see this technique at work.

When one `OptionalExceptions` derivative passes its exceptions flag to another such object, it is only passing a copy. This means that 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 changed the exception throwing flag.

Some of the exceptions MySQL++ can throw are not optional:

- The largest set of non-optional exceptions are those from the Standard C++ Library. For instance, if your code said "`row[21]`" on a row containing only 5 fields, the `std::vector` underlying the row object will throw an exception. (It will, that is, if it conforms to the standard.) You might consider wrapping your program's main loop in a try block catching `std::exception`s, just in case you trigger one of these exceptions.
- ColData 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 pass a bad option value to `Connection::set_option`, it will throw a `BadOption` exception.

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 `ColData` 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 q = conn.query();
q << "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 various string

types, for any type of data that can be converted to MySQL++'s ColData type, and for Specialized SQL Structures. (The next section introduces the SSQLS feature.)

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 q = conn.query();
q << "SELECT * FROM stock WHERE item = " << quote << s;
```

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

3.6. Specialized SQL Structures

Retrieving data

The next example introduces one of the most powerful features of MySQL++: Specialized SQL Structures (SSQLS). This is `examples/custom1.cpp`:

```
#include "stock.h"
#include "util.h"

#include <iostream>
#include <vector>

using namespace std;

int
main(int argc, char *argv[])
{
// Wrap all MySQL++ interactions in one big try block, so any
// errors are handled gracefully.
try {
// Establish the connection to the database server.
mysqlpp::Connection con(mysqlpp::use_exceptions);
if (!connect_to_db(argc, argv, con)) {
return 1;
}

// Retrieve the entire contents of the stock table, and store
// the data in a vector of 'stock' SSQLS structures.
mysqlpp::Query query = con.query();
```

```
query << "select * from stock";
vector<stock> res;
query.storein(res);

// Display the result set
print_stock_header(res.size());
vector<stock>::iterator it;
for (it = res.begin(); it != res.end(); ++it) {
print_stock_row(it->item, it->num, it->weight, it->price,
it->sdate);
}
}
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 many others:

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

#include <string>

// The following is calling a very complex macro which will create
// "struct stock", which has the member variables:
//
//   sql_char item;
//   ...
//   sql_date sdate;
//
// plus methods to help populate the class from a MySQL row.  See the
// SSQLS sections in the user manual for further details.
sql_create_5(stock,
1, 5, // 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, price,
mysqlpp::sql_date, sdate)
```

As you can see, SSQLS is very powerful. It allows you to have a C++ structure paralleling your SQL table structure and use it easily with STL code.

Adding data

SSQLS can also be used to add data to a table. This is `examples/custom2.cpp`:

```
#include "stock.h"
#include "util.h"

#include <iostream>

using namespace std;

int
main(int argc, char *argv[])
{
try {
// Establish the connection to the database server.
mysqlpp::Connection con(mysqlpp::use_exceptions);
if (!connect_to_db(argc, argv, con)) {
return 1;
}

// 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, 1.75, "1998-09-25");

// 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.preview() << endl;

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

// Print the new table.
mysqlpp::Result res;
get_stock_table(query, res);
print_stock_rows(res);
}
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!

There is one subtlety: MySQL++ automatically quotes and escapes the data when building SQL queries using SSQLS structures. It's efficient, too: MySQL++ is smart enough to apply quoting and escaping only for those data types that actually require it.

Because this example modifies the sample database, you may want to run resetdb after running this program.

Modifying data

It almost as easy to modify data with SSQLS. This is `examples/custom3.cpp`:

```
#include "stock.h"
#include "util.h"

#include <iostream>

using namespace std;

int
main(int argc, char *argv[])
{
try {
// Establish the connection to the database server.
mysqlpp::Connection con(mysqlpp::use_exceptions);
if (!connect_to_db(argc, argv, con)) {
return 1;
}

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

// Retrieve the row, throwing an exception if it fails.
mysqlpp::Result 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.at(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.preview() << endl;

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

// Print the new table contents.
get_stock_table(query, res);
print_stock_rows(res);
}
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;
}
```

When you run the example you will notice that in the WHERE clause only the 'item' field is checked for. This is because SSQLS also also less-than-comparable.

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

Less-than-comparable

SSQLS structures can be sorted and stored in STL associative containers as demonstrated in the next example. This is `examples/custom4.cpp`:

```
#include "stock.h"
#include "util.h"

#include <iostream>

using namespace std;

int
main(int argc, char *argv[])
{
try {
// Establish the connection to the database server.
mysqlpp::Connection con(mysqlpp::use_exceptions);
if (!connect_to_db(argc, argv, con)) {
return 1;
}

// 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 custom1.cpp.  It works because
// SSQLS objects are less-than comparable.
mysqlpp::Query query = con.query();
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;
}
```

For more details on the SSQLS feature, see the Specialized SQL Structures chapter.

3.7. C++ Equivalents of SQL Column Types

In MySQL++ version 2.1, the new `sql_types.h` header declares typedefs for all MySQL column types. These typedefs all begin with `sql_` and end with a lowercase version of the standard SQL type name. For instance, the MySQL++ typedef corresponding to `TINYINT UNSIGNED` is `mysqlpp::sql_tinyint_unsigned`. You do not have to use these typedefs; you could use an `unsigned char` here if you wanted to. For that matter, you could use a plain `int` in most cases; MySQL++ is quite tolerant of this sort of thing. The typedefs exist for style reasons, for those who want their C++ code to use the closest equivalent type for any given SQL type.

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`.

3.8. Handling SQL Nulls

There is no equivalent of SQL's null in the standard C++ type system.

The primary distinction is one of type: in SQL, null is a column attribute, which affects whether that column can hold a SQL null. Just like the const keyword in the C++ type system, this effectively doubles the number of SQL data types. To emulate this, MySQL++ provides the Null template to allow the creation of distinct "nullable" versions of existing C++ types. So for example, if you have a TINY INT UNSIGNED column that can have nulls, the proper declaration for MySQL++ would be:

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

Template instantiations are first-class types in the C++ language, on par with any other type. You can use `Null` template instantiations anywhere you'd use the plain version of that type. (You can see a complete list of `Null` template instantiations for all column types that MySQL understands at the top of `lib/type_info.cpp`.)

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;
```

The final aspect of MySQL++'s null handling is that, by default, it will enforce the uniqueness of the SQL null value. If you try to convert a SQL null to a plain C++ data type, MySQL++ will throw a BadNullConversion exception. If you insert a SQL null into a C++ stream, you get "(NULL)". If you don't like this behavior, you can change it, 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 the null type. To relax this distinction, you can instantiate the `Null` template with a different behavior type: NullisZero or NullisBlank. Consider this code:

```
mysqlpp::Null<unsigned char, mysqlpp::NullisZero> myfield;

myfield = mysqlpp::null;
cout << myfield << endl;

int x = myfield;
cout << x << 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 would have thrown a `BadNullConversion` exception.

3.9. Creating Transaction Sets

MySQL++ v2.1 added the Transaction class, which makes it easier to use 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/xaction.cpp` illustrates this:

```
#include "stock.h"
#include "util.h"

#include <transaction.h>

#include <iostream>

using namespace std;

int
main(int argc, char *argv[])
{
try {
// Establish the connection to the database server.
mysqlpp::Connection con(mysqlpp::use_exceptions);
if (!connect_to_db(argc, argv, con)) {
```

```
return 1;
}

// 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
{
mysqlpp::Transaction trans(con);

stock row1("Sauerkraut", 42, 1.2, 0.75, "2006-03-06");
query.insert(row1);
query.execute();
query.reset();

stock row2("Bratwurst", 24, 3.0, 4.99, "2006-03-06");
query.insert(row2);
query.execute();
query.reset();

cout << "\nRows are 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
{
mysqlpp::Transaction trans(con);
cout << "\nNow adding catsup to the database..." << endl;

stock row("Catsup", 3, 3.9, 2.99, "2006-03-06");
query.insert(row);
query.execute();
query.reset();
}
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;
}
```

3.10. 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 ResNSel 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, there's `Query::exec()`, which just returns bool.

If your query does pull data from the database, the simplest option is `store()`. This returns a Result object, which contains an in-memory copy of the result set. The nice thing about this is that `Result` is a sequential container, like `std::vector`, so you can iterate through it forwards and backwards, access elements with subscript notation, etc. There are also the `storein()` methods, which actually put the result set into an STL container of your choice. The downside of these methods is that a sufficiently large result set will give your program memory problems.

For these large result sets, the superior option is a `use()` query. This returns a ResUse object, which is similar to Result, 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 "util.h"

#include <mysql++.h>

#include <iostream>
#include <iomanip>

using namespace std;

int
main(int argc, char *argv[])
{
// Connect to the sample database.
mysqlpp::Connection con(false);
if (!connect_to_db(argc, argv, con)) {
return 1;
}
```

```
// Ask for all rows from the sample stock table set up by resetdb.
// Unlike simple2 example, we don't store result set in memory.
mysqlpp::Query query = con.query();
query << "select * from stock";
mysqlpp::ResUse res = query.use();

// Retreive result rows one by one, and display them.
if (res) {
// Display header
cout.setf(ios::left);
cout << setw(21) << "Item" <<
setw(10) << "Num" <<
setw(10) << "Weight" <<
setw(10) << "Price" <<
"Date" << endl << endl;

// Get each row in result set, and print its contents
mysqlpp::Row row;
while (row = res.fetch_row()) {
cout << setw(20) << row["item"] << ' ' <<
setw(9) << row["num"] << ' ' <<
setw(9) << row["weight"] << ' ' <<
setw(9) << row["price"] << ' ' <<
setw(9) << row["sdate"] <<
endl;
}

return 0;
}
else {
cerr << "Failed to get stock item: " << query.error() << endl;
return 1;
}
}
```

This example does the same thing as `simple2`, only with a "use" query instead of a "store" query. If your program uses exceptions, you should instead look at `examples/usequery.cpp`, which does the same thing as `simple3`, but with exception-awareness.

3.11. Conditional Result Row Handling

`Query::store()` is fine if you really need all the rows the query returns. It sometimes happens that you can't express the full selection criteria in a SQL query. Instead of storing the full result set, then picking over it to find the rows you want to keep, use `Query::store_if()`. This is `examples/store_if.cpp`:

```
#include "util.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 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[])
{
try {
// Connect to the sample database
mysqlpp::Connection con;
if (!connect_to_db(argc, argv, con)) {
return 1;
}

// 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. 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.

3.12. 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 new `Query::for_each()` facility might be just what you need. This is `examples/for_each.cpp`:

```
#include "util.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);
}

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[])
{
try {
// Connect to the sample database
mysqlpp::Connection con;
if (!connect_to_db(argc, argv, con)) {
return 1;
}

// 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.

3.13. 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/fieldinf1.cpp`:

```
#include "util.h"

#include <mysql++.h>

#include <iostream>
#include <iomanip>

using namespace std;
using namespace mysqlpp;

int
main(int argc, char *argv[])
{
try {
Connection con(use_exceptions);
if (!connect_to_db(argc, argv, con)) {
return 1;
}

Query query = con.query();
query << "select * from stock";
cout << "Query: " << query.preview() << endl;

Result res = query.store();
cout << "Records Found: " << res.size() << endl << endl;

cout << "Query Info:\n";
cout.setf(ios::left);

for (unsigned int i = 0; i < res.names().size(); i++) {
cout << setw(2) << i
// this is the name of the field
<< setw(15) << res.names(i).c_str()
// this is the SQL identifier name
// Result::types(unsigned int) returns a mysql_type_info which in many
// ways is like type_info except that it has additional sql type
// information in it. (with one of the methods being sql_name())
<< setw(15) << res.types(i).sql_name()
// this is the C++ identifier name which most closely resembles
// the sql name (its is implementation defined and often not very readable)
<< setw(20) << res.types(i).name()
<< endl;
}

cout << endl;
```

```
if (res.types(0) == typeid(string)) {
// this is demonstrating how a mysql_type_info can be
// compared with a C++ type_info.
cout << "Field 'item' is of an SQL type which most "
"closely resembles\nthe C++ string type\n";
}

if (res.types(1) == typeid(longlong)) {
cout << "Field 'num' is of an SQL type which most "
"closely resembles\nC++ long long int type\n";
}
else if (res.types(1).base_type() == typeid(longlong)) {
// you have to be careful as if it can be null the actual
// type is Null<TYPE> not TYPE.  So you should always use
// the base_type method to get at the underlying type.
// If the type is not null than this base type would be
// the same as its type.
cout << "Field 'num' base type is of an SQL type which "
"most closely\nresembles the C++ long long int type\n";
}
}
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;
}
```

3.14. Dealing with Binary Data

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. Recent releases of MySQL++ do a better job of letting you keep data in forms that don't have this problem, but it's still possible to do it incorrectly. These examples demonstrate correct techniques.

Loading a binary file into a BLOB column

This example shows how to insert binary data into a MySQL table's BLOB column with MySQL++, and also how to get the value of the auto-increment column from the previous insert. (This MySQL feature is usually used to create

unique IDs for rows as they're inserted.) The program requires one command line parameter over that required by the other examples you've seen so far, the path to a JPEG file. This is `examples/load_jpeg.cpp`:

```
#include "util.h"

#include <mysql++.h>

#include <fstream>

using namespace std;
using namespace mysqlpp;

static bool
is_jpeg(const unsigned char* img_data)
{
return (img_data[0] == 0xFF) && (img_data[1] == 0xD8) &&
((memcmp(img_data + 6, "JFIF", 4) == 0) ||
(memcmp(img_data + 6, "Exif", 4) == 0));
}


int
main(int argc, char *argv[])
{
// Assume that the last command line argument is a file.  Try to
// read that file's data into img_data, and check it to see if it
// appears to be a JPEG file.  Bail otherwise.
string img_data;
if ((argc > 1) && (argv[1][0] != '-')) {
ifstream img_file(argv[argc - 1], ios::ate);
if (img_file) {
size_t img_size = img_file.tellg();
if (img_size > 10) {
img_file.seekg(0, ios::beg);
char* img_buffer = new char[img_size];
img_file.read(img_buffer, img_size);
if (is_jpeg((unsigned char*)img_buffer)) {
img_data.assign(img_buffer, img_size);
}
else {
cerr << "File does not appear to be a JPEG!" << endl;
}
delete[] img_buffer;
}
else {
cerr << "File is too short to be a JPEG!" << endl;
}
}
}
if (img_data.empty()) {
print_usage(argv[0], "[jpeg_file]");
return 1;
}
--argc;     // pop filename argument off end of list
```

```
try {
// Establish the connection to the database server.
mysqlpp::Connection con(mysqlpp::use_exceptions);
if (!connect_to_db(argc, argv, con)) {
return 1;
}

// Insert image data into the BLOB column in the images table.
// We're inserting it as an std::string instead of using the raw
// data buffer allocated above because we don't want the data
// treated as a C string, which would truncate the data at the
// first null character.
Query query = con.query();
query << "INSERT INTO images (data) VALUES(\"" <<
mysqlpp::escape << img_data << "\")";
ResNSel res = query.execute();

// If we get here, insertion succeeded
cout << "Inserted \"" << argv[argc] <<
"\" into images table, " << img_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 we're not using one of the MySQL++ types that does automatic escaping and quoting.

Serving images from BLOB column via CGI

This example is also a very short one, considering the function that it performs. It retreives data loaded by `load_jpeg` and prints it out in the form a web server can accept for a CGI call. This is `examples/cgi_jpeg.cpp`:

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

```
using namespace std;
using namespace mysqlpp;

#define IMG_DATABASE    "mysql_cpp_data"
#define IMG_HOST        "localhost"
#define IMG_USER        "root"
#define IMG_PASSWORD    "nunyabinness"

sql_create_2(images,
1, 2,
mysqlpp::sql_int_unsigned, id,
mysqlpp::sql_blob, data)

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

Connection con(use_exceptions);
try {
con.connect(IMG_DATABASE, IMG_HOST, IMG_USER, IMG_PASSWORD);
Query query = con.query();
query << "SELECT * FROM images WHERE id = " << img_id;
ResUse res = query.use();
if (res) {
images img = res.fetch_row();
```

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

return 0;
}
```

You install this in a web server's CGI program directory (usually called `cgi-bin`), 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 table and returns it to the web server, which will send it on to the browser.

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/tquery.cpp`:

```
#include "util.h"

#include <iostream>

using namespace std;

int
main(int argc, char *argv[])
{
try {
// Establish the connection to the database server.
mysqlpp::Connection con(mysqlpp::use_exceptions);
if (!connect_to_db(argc, argv, con)) {
return 1;
}
```

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

// Retrieve an item added by resetdb; it won't be there if
// tquery or custom3 is run since resetdb.
mysqlpp::Result 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();
query << "update stock set item = %0q where item = %1q";
query.parse();
mysqlpp::ResNSel res2 = query.execute("Nuerenberger Bratwurst",
res1.at(0).at(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 quote and escape the item using the MySQL C API function mysql-escape-string if it is a string or char *, or another MySQL-specific type that needs to be quoted.
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
r	Always quote and escape even if it is a number.
R	Always quote but don't escape even if it is a number.

":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:

```
Result 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. Parameter Types and Function Overloads

There are quite a few overloads for each of `Query`'s query execution functions. (`store()`, `use()`, `execute()`...) It's possible to have code that looks like it should work, but which doesn't, because it's calling the wrong overload. For instance:

```
query.storein(my_vector, "1");
query.storein(my_vector, 1);
```

The first one works, and the second does not. The cause is a vestigial second parameter to one of `storein()`'s overloads that's compatible with integers. Being vestigial, it's only getting in the way right now, but we can't fix it until the next major version of the libary, where it will be okay to break the ABI. Until then, we're stuck with it.

If the MySQL server keeps rejecting your template queries, try explicitly casting the parameters to `SQLString`:

```
query.storein(my_vector, SQLString(1));
```

This ensures that your code calls one of the overloads meant to handle template query parameters. I don't recommend doing this habitually, because it will clutter your code. For the most part, MySQL++'s interface is set up to do the right thing. It's just that there are still a few corner cases that can't be fixed until the next time we can redesign the interface.

4.4. 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::def` array. You can refer to the parameters either by position or by name:

```
query.def[1] = "item";
query.def["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.def["field1"] = "item";
query.def["field2"] = "price";
Result res1 = query.store("Hamburger Buns", "item");
Result 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.def["what"] = "Hamburger Buns";
query.def["wheref"] = "item";
query.def["field1"] = "item";
query.def["field2"] = "price";
Result 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.

4.5. 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::def`, 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.def["field1"] = "item";
query.def["field2"] = "price";
Result 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. Because of the extra functionality that this feature builds into these structures, MySQL++ can populate them automatically when retrieving data from the database; with queries returning many records, you can ask MySQL++ to populate an STL container of your SSQLS records with the results. When updating the database, MySQL++ can use SSQLS structures to match existing data, and it can insert SSQLS structures directly into the database.

You define an SSQLS using one of several macros. (These are in the file custom.h, and in the file that it includes, custom-macros.h.) There are a bunch of different macros, for different purposes. The following sections will discuss each macro type separately, beginning with the easiest and most generally useful.

5.1. sql_create

This is the most basic sort of SSQLS declaration:

```
sql_create_5(stock, 1, 0,
string, item,
int, num,
double, weight,
double, price,
mysqlpp::Date, date)
```

This creates a C++ structure called `stock` containing five member variables (`item`, `num`, `weight`, `price` and `date`), along with some constructors and other member functions useful with MySQL++.

One of the generated constructors takes a reference to a mysqlpp::Row object, allowing you to easily populate a vector of stocks like so:

```
vector<stock> result;
query.storein(result);
```

That's all there is to it. The only requirements are that the table structure be compatible with the SSQLS's member variables, and that the fields are in the same order.

The general format of this set of macros 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

`sql_create_`*x* 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 table schema 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_`*x* 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_`*x* 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_5(stock, 1, 2,
string, item,
int, num,
double, weight,
double, price,
mysqlpp::Date, date)

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++. Why does this happen? 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. It is easy to work around this limitation. Using our `stock` example structure, if you wanted comparisons to consider all 5 fields and also be able to initialize all 5 fields at once, you would pass 5 for *`COMPCOUNT`* and 0 for *`SETCOUNT`*. You would still get a 5-parameter initialization constructor and a 5-parameter `set()` function.

5.3. Retrieving a Table Subset

It's not necessary to retrieve an entire table row using SSQLS, as long as the fields you want are grouped together at the start of the table schema. `examples/custom6.cpp` illustrates this:

```
#include "util.h"

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

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

using namespace std;

// To store a subset of a row, we define an SSQLS containing just the
// fields that we will store.  There are complications here that are
// covered in the user manual.
sql_create_1(stock_subset,
1, 0,
string, item)

int
main(int argc, char *argv[])
{
try {
// Establish the connection to the database server.
```

```
mysqlpp::Connection con(mysqlpp::use_exceptions);
if (!connect_to_db(argc, argv, con)) {
return 1;
}

// Retrieve a subset of the stock table, and store the data in
// a vector of 'stock_subset' SSQLS structures.
mysqlpp::Query query = con.query();
query << "select item from stock";
vector<stock_subset> res;
query.storein(res);

// Display the result set
cout << "We have:" << endl;
vector<stock_subset>::iterator it;
for (it = res.begin(); it != res.end(); ++it) {
cout << '\t' << it->item << 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;
}
```

(See the simple1 example in the Tutorial for another way to accomplish the same thing.)

This example illustrates an important point: you could not use the 5-member `stock` structure in this example. The reason is, when you assign a `Row` object to an SSQLS, the function that copies the row's data into the structure expects to see as many fields in the row as are in the SSQLS. Your program will crash when the code tries to access fields beyond those that exist in the `Row` object. The converse is not true, however: if you change the **`SELECT`** statement above so that it retrieves more than one column, the code will still work, because the extra fields in each row will simply be ignored.

Realize that the second and third parameters to `sql_create_1` can't be anything other than 1 and 0, respectively. As discussed above, the second parameter must be at least 1, but since there is only one field in the structure, it cannot be higher than 1. Since the third parameter cannot be equal to the second, only 0 works there.

5.4. Additional Features of Specialized SQL Structures

Up to this point, we haven't been using all of the features in the SSQLS structures we've been generating. We could have used the sql_create_basic_* macros instead, which would have worked just as well for what we've seen so far, and the generated code would have been smaller.

Why is it worth ignoring the "basic" variants of these macros, then? Consider this:

```
query.insert(s);
```

This does exactly what you think it does: it inserts 's' into the database. This is possible because a standard SSQLS has functions that the query object can call to get the list of fields and such, which it uses to build an insert query. `query::update()` and `query::replace()` also rely on this SSQLS feature. A basic SSQLS lacks these functions.

Another feature of standard SSQLSes you might find a use for is changing the table name used in queries. By default, the table in the MySQL database 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";
```

5.5. Harnessing SSQLS Internals

Continuing the discussion in the previous section, there is a further set of methods that the non-"basic" versions of the sql_create macros define 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 `custom*.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, "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,date
Equal list: item = 'Dinner Rolls',num = 75,weight = 0.95, price = 0.97,date = '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/custom5.cpp`:

```
#include "stock.h"
#include "util.h"

#include <iostream>
#include <vector>

using namespace std;

int
main(int argc, char *argv[])
{
try {
mysqlpp::Connection con(mysqlpp::use_exceptions);
if (!connect_to_db(argc, argv, con)) {
return 1;
}
```

```
// Get all the rows in the stock table.
mysqlpp::Query query = con.query();
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.reset();
query << "select * from stock where " <<
res[0].equal_list(" and ", stock_weight, stock_price);

// Display the finished query.
cout << "Custom query:\n" << query.preview() << 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_*x* 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.

5.6. Alternate Creation Methods

If for some reason you want your SSQLS data members to have different names than used in the MySQL database, you can do so like this:

```
sql_create_c_names_5(stock, 1, 5,
string, item, "item",
int, num, "quantity",
double, weight, "weight",
double, price, "price"
mysqlpp::Date, date, "shipment")
```

If you want your SSQLS to have its data members in a different order from those in the MySQL table, you can do it like this:

```
sql_create_c_order_5(stock, 2, 5,
mysqlpp::Date, date, 5,
double, price, 4,
string, item, 1,
int, num, 2,
double, weight, 3)
```

You can combine the custom names and custom ordering like this:

```
sql_create_complete_5(stock, 2, 5,
mysqlpp::date, date, "shipment", 5,
double, price, "price", 4,
string, item, "item", 1,
int, num, "quantity", 2,
double, weight, "weight", 3)
```

All three of these macro types have "basic" variants that work the same way. Again, basic SSQLSes lack the features necessary for automatic insert, update and replace query creation.

5.7. 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 **-I** flag at the top of the script.

5.8. Extending the SSQLS Mechanism

The SSQLS headers — custom.h and custom-macros.h — are automatically generated by the Perl script `custom.pl`. Although it is possible to change this script to get additional functionality, it's usually better to do that through inheritance.

A regular user may find it helpful to change the the limit on the maximum number of SSQLS data members allowed. It's 25 out of the box. A smaller value may speed up compile time, or you may require a higher value because you have more complex tables than that. Simply change the `max_data_members` variable at the top of `custom.pl` and say **make**. The limit for Visual C++ is 31, according to one report. There doesn't seem to be a practical limit with GCC 3.3 at least: I set the limit to 100 and the only thing that happened is that custom-macros.h went from 1.3 MB to 18 MB and the build time for `examples/custom.*` got a lot longer.

5.9. 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 ColData, 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 `ColData` object's contents, without interference.

Because `ColData` derives from `std::string` and C++ strings handle binary data just fine, you might think you can use `std::string` instead of `sql_blob`, but the current design of `ColData` 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 `ColData` or the SSQLS mechanism.

The `sql_blob` typedef may be changed to alias a different type in the future, so using it instead of `ColData` ensures that your code tracks these library changes automatically. Besides, `ColData` 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 staying within the current library interface design.

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

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

using namespace std;
using namespace mysqlpp;

#define IMG_DATABASE    "mysql_cpp_data"
#define IMG_HOST        "localhost"
#define IMG_USER        "root"
#define IMG_PASSWORD    "nunyabinness"

sql_create_2(images,
1, 2,
```

```
mysqlpp::sql_int_unsigned, id,
mysqlpp::sql_blob, data)

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

Connection con(use_exceptions);
try {
con.connect(IMG_DATABASE, IMG_HOST, IMG_USER, IMG_PASSWORD);
Query query = con.query();
query << "SELECT * FROM images WHERE id = " << img_id;
ResUse res = query.use();
if (res) {
images img = res.fetch_row();
cout << "Content-type: image/jpeg" << endl;
cout << "Content-length: " << img.data.length() << "\n\n";
cout << img.data;
}
else {
cout << "Content-type: text/plain" << endl << endl;
cout << "ERROR: No such image with ID " << img_id << endl;
}
}
catch (const BadQuery& er) {
// Handle any query errors
```

```
cout << "Content-type: text/plain" << endl << endl;
cout << "QUERY ERROR: " << er.what() << endl;
return 1;
}
catch (const Exception& er) {
// Catch-all for any other MySQL++ exceptions
cout << "Content-type: text/plain" << endl << endl;
cout << "GENERAL ERROR: " << er.what() << endl;
return 1;
}

return 0;
}
```

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. The subset covering the most economically valuable cases takes two bytes per character, so most Unicode-aware programs deal in 2-byte characters, for efficiency.

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 a reason to do anything more than just pass data along unchanged, 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.) Thus, when explicit support for Unicode was added in MySQL v4.1, they chose to make UTF-8 the native encoding, to preserve backward compatibility with programs that had no Unicode support.

6.2. 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.

6.3. 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. 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.

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, or any "Server" variant of Windows. Windows 95 and its descendants (98, ME, and CE) do not support UCS-2. 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.

6.4. 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. Important Underlying C API Limitations

Since MySQL++ is built on top of the MySQL C API (libmysqlclient), it shares all of its limitations. The following points out some of these limitations that frequently bite newbies. Some of these may be papered over at the MySQL++ layer in future releases, but it's best to write your program as if they were permanent fixtures of the universe.

1. *Only one active query per connection.* This one bites MySQL++ newbies most often in multithreaded programs. If the program has only one Connection object and each thread gets their Query objects from it, it's inevitable that one of those query objects will try to execute while another query is already running on that single connection. The safest course is to have a separate Connection object per thread, and for your code to get Query objects in a thread only from that thread's Connection object. Alternately, you can confine MySQL database access to a single thread.

2. *You must consume all rows from a query before you can start a new query.* This one bites MySQL++ newbies most often when they try code like this:

   ```
   Connection c(...);
   Query q = c.query();
   Result r1 = q.use("select garbage from plink where foobie='tamagotchi'");
   Result r2 = q.use("select blah from bonk where bletch='smurf'");
   ```

   This will fail because a "use" query consumes rows only on demand, so the MySQL server is still keeping information around about the first query when the second one comes in on the connection. When you try the second query, MySQL++ will throw an exception containing an obscure MySQL C API error message about "commands out of sync".

   This is not the only situation where this can happen, but all of these issues boil down to the fact that MySQL requires that certain operations complete before you can start a new one.

3. *The `Result` object must outlive the use of any `Row` objects it returns.* This is because the `Row` objects refer back to the `Result` object that created them for certain data. (Field names, for example.) MySQL does this for efficiency, because there is some information about a row that is the same for all rows in a result set. We could avoid this in MySQL++ by making redundant copies of this data for each row, but that would be quite wasteful.

Beware of some of the more obscure ways this can happen. For example:

```
Connection c(...);
Query q = c.query();
Result res = q.store("...");
Row row = res.at(0);
res = q.store("...");
```

At this point, the `row` variable's contents are likely no longer usable. The program may run, but the row object will use data (field names, etc.) from the second query, not the first.

8. 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, but the ChangeLog 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 ChangeLog.

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

...to something more like this:

```
Query q = con.query();
q << "delete from mytable where myfield=%0";
q.parse();
q.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.

8.2. 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.

9. 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 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.

9.1. 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!

9.2. 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.