MySQL++

3.0.6

Generated by Doxygen 1.5.6

Sat Aug 16 14:51:07 2008

Contents

Chapter 1

MySQL++ Reference Manual

1.1 Getting Started

The best place to get started is the `user manual`. It provides a guide to the example programs and more.

1.2 Major Classes

In MySQL++, the main user-facing classes are **mysqlpp::Connection** (p. 32), **mysqlpp::Query** (p. 126), **mysqlpp::Row** (p. 166), **mysqlpp::StoreQueryResult** (p. 209), and **mysqlpp::UseQueryResult** (p. 247).

In addition, MySQL++ has a mechanism called Specialized SQL Structures (SSQLS), which allow you to create C++ structures that parallel the definition of the tables in your database schema. These let you manipulate the data in your database using native C++ data structures. Programs using this feature often include very little SQL code, because MySQL++ can generate most of what you need automatically when using SSQLSes. There is a whole chapter in the user manual on how to use this feature of the library, plus a section in the user manual's tutorial chapter to introduce it. It's possible to use MySQL++ effectively without using SSQLS, but it sure makes some things a lot easier.

1.3 Major Files

The only two header files your program ever needs to include are mysql++.h, and optionally **custom.h** (p. 265). (The latter implements the SSQLS mechanism.) All of the other files are used within the library only.

1.4 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, which you can subscribe to here: `http://lists.mysql.com/plusplus`

That 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. Questions that are about MySQL++ usage may well be ignored if you send them to our personal email accounts. Those of us still active in MySQL++ development monitor the mailing list, so you aren't getting any extra "coverage" by sending messages to those addresses in addition to the mailing list.

1.5 Licensing

MySQL++ is licensed under the GNU Lesser General Public License, which you should have received with the distribution package in a file called "LGPL" or "LICENSE". You can also view it here: `http://www.gnu.org/licenses/lgpl.html` or receive a copy by writing to Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

Chapter 2

Class Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Class Documentation

5.1 AutoFlag$<$ T $>$ Class Template Reference

A template for setting a flag on a variable as long as the object that set it is in scope. Flag resets when object goes out of scope. Works on anything that looks like bool.

`#include <autoflag.h>`

Collaboration diagram for AutoFlag$<$ T $>$:

Public Member Functions

- **AutoFlag** (T &ref)

  Constructor: sets ref to true.

- ∼**AutoFlag** ()

  Destructor: sets referent passed to ctor to false.

5.1.1 Detailed Description

template<class T = bool> class AutoFlag< T >

A template for setting a flag on a variable as long as the object that set it is in scope. Flag resets when object goes out of scope. Works on anything that looks like bool.

The documentation for this class was generated from the following file:

- **autoflag.h**

5.2 mysqlpp::BadConversion Class Reference

Exception (p. 84) thrown when a bad type conversion is attempted.

`#include <exceptions.h>`

Inheritance diagram for mysqlpp::BadConversion:

Collaboration diagram for mysqlpp::BadConversion:

Public Member Functions

- **BadConversion** (const char ∗tn, const char ∗d, size_t r, size_t a)

  Create exception object, building error string dynamically.

- **BadConversion** (const std::string &w, const char ∗tn, const char ∗d, size_t r, size_t a)

  Create exception object, given completed error string.

- **BadConversion** (const char ∗w="")

  Create exception object, with error string only.

- ∼**BadConversion** () throw ()

  Destroy exception.

Public Attributes

- const char ∗ **type_name**

  name of type we tried to convert to

- std::string **data**

  string form of data we tried to convert

- size_t **retrieved**

  documentation needed!

- size_t **actual_size**

  documentation needed!

5.2.1 Detailed Description

Exception (p. 84) thrown when a bad type conversion is attempted.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 mysqlpp::BadConversion::BadConversion (const char ∗ *tn*, const char ∗ *d*, size_t *r*, size_t *a*) `[inline]`

Create exception object, building error string dynamically.

Parameters:

tn type name we tried to convert to

d string form of data we tried to convert

r ??

a ??

5.2.2.2 mysqlpp::BadConversion::BadConversion (const std::string & *w*, const char ∗ *tn*, const char ∗ *d*, size_t *r*, size_t *a*) `[inline]`

Create exception object, given completed error string.

Parameters:

w the "what" error string

tn type name we tried to convert to

d string form of data we tried to convert

r ??

a ??

5.2.2.3 mysqlpp::BadConversion::BadConversion (const char $*$ *w* = "") `[inline, explicit]`

Create exception object, with error string only.

Parameters:

w the "what" error string

All other data members are initialize to default values

The documentation for this class was generated from the following file:

- **exceptions.h**

5.3 mysqlpp::BadFieldName Class Reference

Exception (p. 84) thrown when a requested named field doesn't exist.

`#include <exceptions.h>`

Inheritance diagram for mysqlpp::BadFieldName:

Collaboration diagram for mysqlpp::BadFieldName:

Public Member Functions

- **BadFieldName** (const char ∗bad_field)

  Create exception object.

- ∼**BadFieldName** () throw ()

  Destroy exception.

5.3.1 Detailed Description

Exception (p. 84) thrown when a requested named field doesn't exist.

Thrown by Row::lookup_by_name() when you pass a field name that isn't in the result set.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 mysqlpp::BadFieldName::BadFieldName (const char * *bad_field*) `[inline, explicit]`

Create exception object.

Parameters:

bad_field name of field the database server didn't like

The documentation for this class was generated from the following file:

- **exceptions.h**

5.4 mysqlpp::BadOption Class Reference

Exception (p. 84) thrown when you pass an unrecognized option to **Connection::set_-option()** (p. 40).

`#include <exceptions.h>`

Inheritance diagram for mysqlpp::BadOption:

Collaboration diagram for mysqlpp::BadOption:

Public Member Functions

- **BadOption** (const char ∗w, const std::type_info &ti)

  Create exception object, taking C string.

- **BadOption** (const std::string &w, const std::type_info &ti)

  Create exception object, taking C++ string.

- const std::type_info & **what_option** () const

  Return type information about the option that failed.

5.4.1 Detailed Description

Exception (p. 84) thrown when you pass an unrecognized option to **Connection::set_-option()** (p. 40).

5.4.2 Member Function Documentation

5.4.2.1 const std::type_info& mysqlpp::BadOption::what_option () const `[inline]`

Return type information about the option that failed.

Because each option has its own C++ type, this lets you distinguish among **BadOption** (p. 20) exceptions programmatically.

The documentation for this class was generated from the following file:

- **exceptions.h**

5.5 mysqlpp::BadParamCount Class Reference

Exception (p. 84) thrown when not enough query parameters are provided.

`#include <exceptions.h>`

Inheritance diagram for mysqlpp::BadParamCount:

Collaboration diagram for mysqlpp::BadParamCount:

Public Member Functions

- **BadParamCount** (const char ∗w="")

  Create exception object.
- ∼**BadParamCount** () throw ()

  Destroy exception.

5.5.1 Detailed Description

Exception (p. 84) thrown when not enough query parameters are provided.

This is used in handling template queries.

The documentation for this class was generated from the following file:

- **exceptions.h**

5.6 mysqlpp::BadQuery Class Reference

Exception (p. 84) thrown when the database server encounters a problem while processing your query.

```
#include <exceptions.h>
```

Inheritance diagram for mysqlpp::BadQuery:

Collaboration diagram for mysqlpp::BadQuery:

Public Member Functions

- **BadQuery** (const char ∗w="", int e=0)

  Create exception object.

- **BadQuery** (const std::string &w, int e=0)

  Create exception object.

- int **errnum** () const

  *Return the error number corresponding to the error message returned by **what()** (p. 85).*

5.6.1 Detailed Description

Exception (p. 84) thrown when the database server encounters a problem while processing your query.

Unlike most other MySQL++ exceptions, which carry just an error message, this type carries an error number to preserve **Connection::errnum()** (p. 33)'s return value at the point the exception is thrown. We do this because when using the **Transaction** (p. 237) class, the rollback process that occurs during stack unwinding issues a query to the database server, overwriting the error value. This rollback should always succeed, so this effect can fool code that relies on **Connection::errnum()** (p. 33) into believing that there was no error.

Beware that in older versions of MySQL++, this was effectively the generic exception type. (This is most especially true in v1.7.x, but it continued to a lesser extent through the v2.x series.) When converting old code to new versions of MySQL++, it's therefore possible to get seemingly "new" exceptions thrown, which could crash your program if you don't also catch a more generic type like **mysqlpp::Exception** (p. 84) or std::exception.

5.6.2 Constructor & Destructor Documentation

5.6.2.1 mysqlpp::BadQuery::BadQuery (const char ∗ *w* = "", int *e* = 0) `[inline, explicit]`

Create exception object.

Parameters:

- ***w*** explanation for why the exception was thrown
- ***e*** the error number from the underlying database API

5.6.2.2 mysqlpp::BadQuery::BadQuery (const std::string & *w*, int *e* = 0) `[inline, explicit]`

Create exception object.

Parameters:

- ***w*** explanation for why the exception was thrown
- ***e*** the error number from the underlying database API

5.6.3 Member Function Documentation

5.6.3.1 int mysqlpp::BadQuery::errnum () const `[inline]`

Return the error number corresponding to the error message returned by **what()** (p. 85).

This may return the same value as **Connection::errnum()** (p. 33), but not always. See the overview documentation for this class for the reason for the difference.

The documentation for this class was generated from the following file:

- **exceptions.h**

5.7 mysqlpp::BeecryptMutex Class Reference

Wrapper around platform-specific mutexes.

```
#include <beemutex.h>
```

Public Member Functions

- **BeecryptMutex** () throw (MutexFailed)

  Create the mutex object.

- ∼**BeecryptMutex** ()

  Destroy the mutex.

- void **lock** () throw (MutexFailed)

  Acquire the mutex, blocking if it can't be acquired immediately.

- bool **trylock** () throw (MutexFailed)

  Acquire the mutex immediately and return true, or return false if it would have to block to acquire the mutex.

- void **unlock** () throw (MutexFailed)

  Release the mutex.

5.7.1 Detailed Description

Wrapper around platform-specific mutexes.

This class is only intended to be used within the library. We don't really want to support this as a general purpose class. If it works for you as-is, that's great, we won't try to stop you. But if you run into a problem that doesn't affect MySQL++ itself, we're not likely to bother enhancing this class to fix the problem.

5.7.2 Constructor & Destructor Documentation

5.7.2.1 mysqlpp::BeecryptMutex::BeecryptMutex () throw (MutexFailed)

Create the mutex object.

Throws a **MutexFailed** (p. 101) exception if we can't acquire the lock for some reason. The exception contains a message saying why.

5.7.2.2 mysqlpp::BeecryptMutex::∼BeecryptMutex ()

Destroy the mutex.

Failures are quietly ignored.

The documentation for this class was generated from the following files:

- **beemutex.h**
- beemutex.cpp

5.8 Comparable< T > Class Template Reference

Mix-in that gives its subclass a full set of comparison operators.

`#include <comparable.h>`

Inheritance diagram for Comparable< T >:

Public Member Functions

- bool **operator==** (const T &other) const

  Returns true if "other" is equal to this object.

- bool **operator!=** (const T &other) const

  Returns true if "other" is not equal to this object.

- bool **operator<** (const T &other) const

  Returns true if "other" is less than this object.

- bool **operator<=** (const T &other) const

  Returns true if "other" is less than or equal to this object.

- bool **operator>** (const T &other) const

  Returns true if "other" is greater than this object.

- bool **operator>=** (const T &other) const

  Returns true if "other" is greater than or equal to this object.

Protected Member Functions

- virtual ~**Comparable** ()

  Destroy object.

- virtual int **compare** (const T &other) const =0

  Compare this object to another of the same type.

5.8.1 Detailed Description

template<class T> class Comparable< T >

Mix-in that gives its subclass a full set of comparison operators.

Simply by inheriting publically from this and implementing **compare()** (p. 29), the subclass gains a full set of comparison operators, because all of the operators are implemented in terms of **compare()** (p. 29).

5.8.2 Constructor & Destructor Documentation

5.8.2.1 template<class T> virtual Comparable< T >::~Comparable () `[inline, protected, virtual]`

Destroy object.

This class has nothing to destroy, but declaring the dtor virtual placates some compilers set to high warning levels. Protecting it ensures you can't delete subclasses through base class pointers, which makes no sense because this class isn't made for polymorphism. It's just a mixin.

5.8.3 Member Function Documentation

5.8.3.1 template<class T> virtual int Comparable< T >::compare (const T & *other*) const `[protected, pure virtual]`

Compare this object to another of the same type.

Returns < 0 if this object is "before" the other, 0 of they are equal, and > 0 if this object is "after" the other.

Referenced by Comparable< mysqlpp::DateTime >::operator!=(), Comparable< mysqlpp::DateTime >::operator<(), Comparable< mysqlpp::DateTime >::operator<=(), Comparable< mysqlpp::DateTime >::operator==(), Comparable<

mysqlpp::DateTime >::operator>(), and Comparable< mysqlpp::DateTime >::operator>=().

The documentation for this class was generated from the following file:

- **comparable.h**

5.9 mysqlpp::CompressOption Class Reference

Enable data compression on the connection.

```
#include <options.h>
```

Inheritance diagram for mysqlpp::CompressOption:

Collaboration diagram for mysqlpp::CompressOption:

5.9.1 Detailed Description

Enable data compression on the connection.

The documentation for this class was generated from the following file:

- **options.h**

5.10 mysqlpp::Connection Class Reference

Manages the connection to the database server.

`#include <connection.h>`

Inheritance diagram for mysqlpp::Connection:

Collaboration diagram for mysqlpp::Connection:

Public Member Functions

- **Connection** (bool te=true)

  Create object without connecting to the database server.

- **Connection** (const char ∗db, const char ∗server=0, const char ∗user=0, const char ∗password=0, unsigned int port=0)

  Create object and connect to database server in one step.

- **Connection** (const **Connection** &other)

  Establish a new connection using the same parameters as an existing connection.

- virtual ∼**Connection** ()

  Destroy object.

- std::string **client_version** () const

  Get version of library underpinning the current database driver.

- virtual bool **connect** (const char ∗db=0, const char ∗server=0, const char ∗user=0, const char ∗password=0, unsigned int port=0)

  Connect to database after object is created.

- bool **connected** () const

  Returns true if connection was established successfully.

- ulonglong **count_rows** (const std::string &table)

  Returns the number of rows in a table.

- bool **create_db** (const std::string &db)

  Ask the database server to create a database.

- void **disconnect** ()

  Drop the connection to the database server.

- **DBDriver** ∗ **driver** ()

  Returns a reference to the current database driver.

- bool **drop_db** (const std::string &db)

  Asks the database server to drop (destroy) a database.

- int **errnum** ()

  Return last error number associated with this connection.

- const char ∗ **error** () const

  Return error message for last error associated with this connection.

- std::string **ipc_info** () const

  Get information about the IPC connection to the database server.

- bool **kill** (unsigned long tid) const

  Kill a database server thread.

- **operator private_bool_type** () const

  Test whether any error has occurred within the object.

- **Connection** & **operator=** (const **Connection** &rhs)

  Copy an existing ***Connection*** (p. *32) object's state into this object.*

- bool **ping** ()

  "Pings" the database server

- int **protocol_version** () const

  Returns version number of the protocol the database driver uses to communicate with the server.

- **Query query** (const char ∗qstr=0)

  Return a new query object.

- **Query query** (const std::string &qstr)

  Return a new query object.

- bool **select_db** (const std::string &db)

  Change to a different database managed by the database server we are connected to.

- std::string **server_version** () const

  Get the database server's version string.

- bool **set_option** (**Option** ∗o)

  Sets a connection option.

- bool **shutdown** ()

  Ask database server to shut down.

- std::string **server_status** () const

  Returns information about database server's status.

- unsigned long **thread_id** ()

  Returns the database server's thread ID for this connection.

Static Public Member Functions

- static bool **thread_aware** ()

  Returns true if both MySQL++ and database driver we're using were compiled with thread awareness.

- static void **thread_end** ()

  Tells the underlying database driver that this thread is done using the library.

- static bool **thread_start** ()

  Tells the underlying database driver that the current thread is now using its services.

Protected Member Functions

- void **build_error_message** (const char ∗core)

  Build an error message in the standard form used whenever one of the methods can't succeed because we're not connected to the database server.

- void **copy** (const **Connection** &other)

  Establish a new connection as a copy of an existing one.

- bool **parse_ipc_method** (const char ∗server, std::string &host, unsigned int &port, std::string &socket_name)

  *Extract elements from the server parameter in formats suitable for passing to **DBDriver::connect()** (p. 67).*

Protected Attributes

- std::string **error_message_**

  MySQL++ specific error, if any.

5.10.1 Detailed Description

Manages the connection to the database server.

This class is a thick wrapper around **DBDriver** (p. 62), adding high-level error handling, utility functions, and abstraction away from underlying C API details.

5.10.2 Constructor & Destructor Documentation

5.10.2.1 mysqlpp::Connection::Connection (bool *te* = `true`)

Create object without connecting to the database server.

Parameters:

te if true, exceptions are thrown on errors

5.10.2.2 mysqlpp::Connection::Connection (const char ∗ *db*, const char ∗ *server* = `0`, const char ∗ *user* = `0`, const char ∗ *password* = `0`, unsigned int *port* = `0`)

Create object and connect to database server in one step.

This constructor allows you to most fully specify the options used when connecting to the database server.

Parameters:

db name of database to select upon connection

server specifies the IPC method and parameters for contacting the server; see below for details

user user name to log in under, or 0 to use the user name this program is running under

password password to use when logging in

port TCP port number database server is listening on, or 0 to use default value; note that you may also give this as part of the `server` parameter

The server parameter can be any of several different forms:

- **0**: Let the database driver decide how to connect; usually some sort of localhost IPC method.
- **"."**: On Windows, this means named pipes, if the server supports it
- **"/some/domain/socket/path"**: If the passed string doesn't match one of the previous alternatives and we're on a system that supports Unix domain sockets, MySQL++ will test it to see if it names one, and use it if we have permission.
- **"host.name.or.ip:port"**: If the previous test fails, or if the system doesn't support Unix domain sockets at all, it assumes the string is some kind of network address, optionally followed by a colon and port. The name can be in dotted quad form, a host name, or a domain name. The port can either be a TCP/IP port number or a symbolic service name. If a port or service name is given here and a nonzero value is passed for the `port` parameter, the latter takes precedence.

References connect().

5.10.2.3 mysqlpp::Connection::Connection (const Connection & *other*)

Establish a new connection using the same parameters as an existing connection.

Parameters:

other existing **Connection** (p. 32) object

References copy().

5.10.3 Member Function Documentation

5.10.3.1 bool mysqlpp::Connection::connect (const char * *db* = `0`, const char * *server* = `0`, const char * *user* = `0`, const char * *password* = `0`, unsigned int *port* = `0`) `[virtual]`

Connect to database after object is created.

It's better to use the connect-on-create constructor if you can. See its documentation for the meaning of these parameters.

If you call this method on an object that is already connected to a database server, the previous connection is dropped and a new connection is established.

References mysqlpp::DBDriver::connect(), errnum(), error(), error_message_, parse_-ipc_method(), and mysqlpp::OptionalExceptions::throw_exceptions().

Referenced by mysqlpp::WindowsNamedPipeConnection::connect(), mysqlpp::UnixDomainSocketConnection::connect(), mysqlpp::TCPConnection::connect(), and Connection().

5.10.3.2 bool mysqlpp::Connection::connected () const

Returns true if connection was established successfully.

Returns:

true if connection was established successfully

References mysqlpp::DBDriver::connected().

Referenced by ping(), select_db(), and shutdown().

5.10.3.3 ulonglong mysqlpp::Connection::count_rows (const std::string & *table*)

Returns the number of rows in a table.

Parameters:

table name of table whose rows you want counted

This is syntactic sugar for a `SELECT COUNT(*) FROM table` SQL query.

References error_message_, mysqlpp::Query::store(), and mysqlpp::OptionalExceptions::throw_exceptions().

5.10.3.4 bool mysqlpp::Connection::create_db (const std::string & *db*)

Ask the database server to create a database.

Parameters:

db name of database to create

Returns:

true if database was created successfully

References error_message_, mysqlpp::Query::exec(), and mysqlpp::OptionalExceptions::throw_exceptions().

5.10.3.5 bool mysqlpp::Connection::drop_db (const std::string & *db*)

Asks the database server to drop (destroy) a database.

Parameters:

db name of database to destroy

Returns:

true if database was dropped successfully

References error_message_, mysqlpp::Query::exec(), and mysqlpp::OptionalExceptions::throw_exceptions().

5.10.3.6 const char ∗ mysqlpp::Connection::error () const

Return error message for last error associated with this connection.

Returns either a MySQL++-specific error message if one exists, or one from the current database driver otherwise.

References mysqlpp::DBDriver::error(), and error_message_.

Referenced by connect(), mysqlpp::Query::error(), select_db(), and shutdown().

5.10.3.7 std::string mysqlpp::Connection::ipc_info () const

Get information about the IPC connection to the database server.

String (p. 211) contains info about type of connection (e.g. TCP/IP, named pipe, Unix socket...) and the server hostname.

References mysqlpp::DBDriver::ipc_info().

5.10.3.8 bool mysqlpp::Connection::kill (unsigned long *tid*) const

Kill a database server thread.

Parameters:

tid ID of thread to kill

See also:

thread_id() (p. 41)

References error_message_, and mysqlpp::DBDriver::kill().

5.10.3.9 mysqlpp::Connection::operator private_bool_type () const `[inline]`

Test whether any error has occurred within the object.

Allows the object to be used in bool context, like this:

```
Connection conn;
.... use conn
if (conn) {
    ... nothing bad has happened since last successful use
}
else {
    ... some error has occurred
}
```

Prior to MySQL++ v3, the object was always falsy when we weren't connected. Now a true return simply indicates a lack of errors. If you've been using this to test for whether the connection is still up, you need to call **connected()** (p. 37) instead.

5.10.3.10 bool mysqlpp::Connection::ping ()

"Pings" the database server

Return values:

true if server is responding

false if either we already know the connection is down and cannot re-establish it, or if the server did not respond to the ping and we could not re-establish the connection.

References build_error_message(), connected(), error_message_, and mysqlpp::DBDriver::ping().

5.10.3.11 Query mysqlpp::Connection::query (const char ∗ *qstr* = 0)

Return a new query object.

The returned query object is tied to this connection object, so when you call a method like **execute()** (p. 137) on that object, the query is sent to the server this object is connected to.

Parameters:

qstr an optional query string for populating the new **Query** (p. 126) object

References mysqlpp::OptionalExceptions::throw_exceptions().

Referenced by mysqlpp::Transaction::commit(), query(), mysqlpp::Transaction::rollback(), and mysqlpp::Transaction::Transaction().

5.10.3.12 Query mysqlpp::Connection::query (const std::string & *qstr*)

Return a new query object.

Parameters:

qstr initial query string

References query().

5.10.3.13 bool mysqlpp::Connection::select_db (const std::string & *db*)

Change to a different database managed by the database server we are connected to.

Parameters:

db database to switch to

Return values:

true if we changed databases successfully

References build_error_message(), connected(), errnum(), error(), error_message_-, mysqlpp::DBDriver::select_db(), and mysqlpp::OptionalExceptions::throw_-exceptions().

5.10.3.14 bool mysqlpp::Connection::set_option (Option ∗ *o*)

Sets a connection option.

Parameters:

o pointer to any derivative of **Option** (p. 121) allocated on the heap

Objects passed to this method and successfully set will be released when this **Connection** (p. 32) object is destroyed. If an error occurs while setting the option the object will be deleted immediately.

Because there are so many **Option** (p. 121) subclasses, the actual effect of this function has a wide range. This mechanism abstracts away many things that are unrelated down at the database driver level, hiding them behind a coherent, type-safe interface.

The rules about which options can be set, when, are up to the underlying database driver. Some must be set before the connection is established because they can only be used during that connection setup process. Others can be set at any time after the connection comes up. If you get it wrong, you'll get a **BadOption** (p. 20) exception.

Return values:

true if option was successfully set

References error_message_, mysqlpp::DBDriver::set_option(), and mysqlpp::OptionalExceptions::throw_exceptions().

5.10.3.15 unsigned long mysqlpp::Connection::thread_id ()

Returns the database server's thread ID for this connection.

This has nothing to do with threading on the client side. The only thing you can do with this value is pass it to **kill()** (p. 39).

References mysqlpp::DBDriver::thread_id().

5.10.3.16 bool mysqlpp::Connection::thread_start () `[static]`

Tells the underlying database driver that the current thread is now using its services.

It's not necessary to call this from the thread that creates the connection as it's done automatically. This method exists for times when multiple threads may use this object; it allows the underlying database driver to set up any per-thread data structures it needs.

The MySQL++ user manual's `chapter on threads` details two major strategies for dealing with connections in the face of threads. The Connection-per-thread option frees you from ever having to call this method. The other documented strategy is to use **ConnectionPool** (p. 45), which opens the possibility for one thread to create a connection that another uses, so you do need to call this method in that case, or with any other similar strategy.

Return values:

True if there was no problem

References mysqlpp::DBDriver::thread_start().

5.10.3.17 void mysqlpp::Connection::copy (const Connection & *other*) `[protected]`

Establish a new connection as a copy of an existing one.

Parameters:

other the connection to copy

References mysqlpp::DBDriver::copy(), driver_, error_-message_, mysqlpp::OptionalExceptions::set_exceptions(), and mysqlpp::OptionalExceptions::throw_exceptions().

Referenced by Connection(), and operator=().

The documentation for this class was generated from the following files:

- **connection.h**
- connection.cpp

5.11 mysqlpp::ConnectionFailed Class Reference

Exception (p. 84) thrown when there is a problem related to the database server connection.

```
#include <exceptions.h>
```

Inheritance diagram for mysqlpp::ConnectionFailed:

Collaboration diagram for mysqlpp::ConnectionFailed:

Public Member Functions

- **ConnectionFailed** (const char ∗w="", int e=0)

  Create exception object.

- int **errnum** () const

  Return the error number corresponding to the error message returned by ***what()*** *(p. 85), if any.*

5.11.1 Detailed Description

Exception (p. 84) thrown when there is a problem related to the database server connection.

This is thrown not just on making the connection, but also on shutdown and when calling certain of Connection's methods that require a connection when there isn't one.

5.11.2 Constructor & Destructor Documentation

5.11.2.1 mysqlpp::ConnectionFailed::ConnectionFailed (const char ∗ *w* = "", int *e* = 0) `[inline, explicit]`

Create exception object.

Parameters:

- ***w*** explanation for why the exception was thrown
- ***e*** the error number from the underlying database API

5.11.3 Member Function Documentation

5.11.3.1 int mysqlpp::ConnectionFailed::errnum () const `[inline]`

Return the error number corresponding to the error message returned by **what()** (p. 85), if any.

If the error number is 0, it means that the error message doesn't come from the underlying database API, but rather from MySQL++ itself. This happens when an error condition is detected up at this higher level instead of letting the underlying database API do it.

The documentation for this class was generated from the following file:

- **exceptions.h**

5.12 mysqlpp::ConnectionPool Class Reference

Manages a pool of connections for programs that need more than one **Connection** (p. 32) object at a time, but can't predict how many they need in advance.

`#include <cpool.h>`

Collaboration diagram for mysqlpp::ConnectionPool:

Public Member Functions

- **ConnectionPool** ()

    Create empty pool.

- virtual ∼**ConnectionPool** ()

    Destroy object.

- bool **empty** () const

    Returns true if pool is empty.

- virtual **Connection** ∗ **grab** ()

    Grab a free connection from the pool.

- virtual void **release** (const **Connection** ∗pc)

    Return a connection to the pool.

- void **shrink** ()

    Remove all unused connections from the pool.

Protected Member Functions

- void **clear** (bool all=true)

    Drains the pool, freeing all allocated memory.

- virtual **Connection** ∗ **create** ()=0

Create a new connection.

- virtual void **destroy** (**Connection** ∗)=0

  Destroy a connection.

- virtual unsigned int **max_idle_time** ()=0

  Returns the maximum number of seconds a connection is able to remain idle before it is dropped.

- size_t **size** () const

  Returns the current size of the internal connection pool.

Classes

- struct **ConnectionInfo**

5.12.1 Detailed Description

Manages a pool of connections for programs that need more than one **Connection** (p. 32) object at a time, but can't predict how many they need in advance.

This class is useful in programs that need to make multiple simultaneous queries on the database; this requires multiple **Connection** (p. 32) objects due to a hard limitation of the underlying C API. **Connection** (p. 32) pools are most useful in multithreaded programs, but it can be helpful to have one in a single-threaded program as well. Sometimes it's necessary to get more data from the server while in the middle of processing data from an earlier query; this requires multiple connections. Whether you use a pool or manage connections yourself is up to you, but realize that this class takes care of a lot of subtle details for you that aren't obvious.

The pool's policy for connection reuse is to always return the *most* recently used connection that's not being used right now. This ensures that excess connections don't hang around any longer than they must. If the pool were to return the *least* recently used connection, it would be likely to result in a large pool of sparsely used connections because we'd keep resetting the last-used time of whichever connection is least recently used at that moment.

5.12.2 Constructor & Destructor Documentation

5.12.2.1 virtual mysqlpp::ConnectionPool::∼ConnectionPool () `[inline, virtual]`

Destroy object.

If the pool raises an assertion on destruction, it means our subclass isn't calling **clear()** (p. 47) in its dtor as it should.

5.12.3 Member Function Documentation

5.12.3.1 Connection $*$ mysqlpp::ConnectionPool::grab () `[virtual]`

Grab a free connection from the pool.

This method creates a new connection if an unused one doesn't exist, and destroys any that have remained unused for too long. If there is more than one free connection, we return the most recently used one; this allows older connections to die off over time when the caller's need for connections decreases.

Do not delete the returned pointer. This object manages the lifetime of connection objects it creates.

Return values:

a pointer to the connection

References create().

5.12.3.2 void mysqlpp::ConnectionPool::release (const Connection $*$ *pc*) `[virtual]`

Return a connection to the pool.

Marks the connection as no longer in use.

The pool updates the last-used time of a connection only on release, on the assumption that it was used just prior. There's nothing forcing you to do it this way: your code is free to delay releasing idle connections as long as it likes. You want to avoid this because it will make the pool perform poorly; if it doesn't know approximately how long a connection has really been idle, it can't make good judgements about when to remove it from the pool.

5.12.3.3 void mysqlpp::ConnectionPool::clear (bool *all* = `true`) `[protected]`

Drains the pool, freeing all allocated memory.

A derived class must call this in its dtor to avoid leaking all **Connection** (p. 32) objects still in existence. We can't do it up at this level because this class's dtor can't call our subclass's **destroy()** (p. 48) method.

Parameters:

all if true, remove all connections, even those in use

References destroy().

5.12.3.4 virtual Connection∗ mysqlpp::ConnectionPool::create () `[protected, pure virtual]`

Create a new connection.

Subclasses must override this.

Essentially, this method lets your code tell **ConnectionPool** (p. 45) what server to connect to, what login parameters to use, what connection options to enable, etc. **ConnectionPool** (p. 45) can't know any of this without your help.

Return values:

A connected **Connection** (p. 32) object

Referenced by grab().

5.12.3.5 virtual void mysqlpp::ConnectionPool::destroy (Connection ∗) `[protected, pure virtual]`

Destroy a connection.

Subclasses must override this.

This is for destroying the objects returned by **create()** (p. 48). Because we can't know what the derived class did to create the connection we can't reliably know how to destroy it.

Referenced by clear().

5.12.3.6 virtual unsigned int mysqlpp::ConnectionPool::max_idle_time () `[protected, pure virtual]`

Returns the maximum number of seconds a connection is able to remain idle before it is dropped.

Subclasses must override this as it encodes a policy issue, something that MySQL++ can't declare by fiat.

Return values:

number of seconds before an idle connection is destroyed due to lack of use

The documentation for this class was generated from the following files:

- **cpool.h**
- cpool.cpp

5.13 mysqlpp::ConnectTimeoutOption Class Reference

Change **Connection::connect()** (p. 37) default timeout.

`#include <options.h>`

Inheritance diagram for mysqlpp::ConnectTimeoutOption:

Collaboration diagram for mysqlpp::ConnectTimeoutOption:

5.13.1 Detailed Description

Change **Connection::connect()** (p. 37) default timeout.

The documentation for this class was generated from the following file:

- **options.h**

5.14 mysqlpp::DataOption< T > Class Template Reference

Define abstract interface for all ∗Options that take a lone scalar as an argument.

`#include <options.h>`

Inheritance diagram for mysqlpp::DataOption< T >:

Collaboration diagram for mysqlpp::DataOption< T >:

Public Types

- typedef T **ArgType**

  Alias for template param.

Protected Member Functions

- **DataOption** (const T &arg)

  Construct object.

Protected Attributes

- T **arg_**

  The argument value.

5.14.1 Detailed Description

template<typename T> class mysqlpp::DataOption< T >

Define abstract interface for all ∗Options that take a lone scalar as an argument.

The documentation for this class was generated from the following file:

- **options.h**

5.15 mysqlpp::Date Class Reference

C++ form of SQL's DATE type.

`#include <datetime.h>`

Inheritance diagram for mysqlpp::Date:

Collaboration diagram for mysqlpp::Date:

Public Member Functions

- **Date** ()

  Default constructor.

- **Date** (unsigned short y, unsigned char m, unsigned char d)

  Initialize object.

- **Date** (const **Date** &other)

  *Initialize object as a copy of another **Date** (p. 53).*

- **Date** (const **DateTime** &other)

  Initialize object from date part of date/time object.

- **Date** (const char ∗str)

  Initialize object from a C string containing a date.

- template<class Str>
  Date (const Str &str)

  Initialize object from a C++ string containing a date.

- **Date** (time_t t)

  Initialize object from a `time_t`*.*

- int **compare** (const **Date** &other) const

  Compare this date to another.

- const char ∗ **convert** (const char ∗)

  Parse a SQL date string into this object.

- unsigned char **day** () const

  Get the date's day part, 1-31.

- void **day** (unsigned char d)

  Change the date's day part, 1-31.

- unsigned char **month** () const

  Get the date's month part, 1-12.

- void **month** (unsigned char m)

  Change the date's month part, 1-12.

- **operator std::string** () const

  Convert to std::string.

- **operator time_t** () const

  Convert to time_t.

- std::string **str** () const

  Return our value in std::string form.

- unsigned short **year** () const

  Get the date's year part.

- void **year** (unsigned short y)

  Change the date's year part.

5.15.1 Detailed Description

C++ form of SQL's DATE type.

Objects of this class can be inserted into streams, and initialized from SQL DATE strings.

5.15.2 Constructor & Destructor Documentation

5.15.2.1 mysqlpp::Date::Date (const char ∗ *str*) `[inline, explicit]`

Initialize object from a C string containing a date.

String (p. 211) must be in the YYYY-MM-DD format. It doesn't have to be zero-padded.

5.15.2.2 template<class Str> mysqlpp::Date::Date (const Str & *str*) `[inline, explicit]`

Initialize object from a C++ string containing a date.

This works with any stringish class that declares a c_str() member function: std::string, **mysqlpp::String** (p. 211)...

See also:

Date(const char∗) (p. 55)

5.15.2.3 mysqlpp::Date::Date (time_t *t*) `[explicit]`

Initialize object from a `time_t`.

Naturally, we throw away the "time" part of the `time_t`. If you need to keep it, you want to use **DateTime** (p. 57) instead.

5.15.3 Member Function Documentation

5.15.3.1 int mysqlpp::Date::compare (const Date & *other*) const

Compare this date to another.

Returns <0 if this date is before the other, 0 of they are equal, and >0 if this date is after the other.

References day_, month_, and year_.

5.15.3.2 mysqlpp::Date::operator time_t () const

Convert to time_t.

The "time" part of the `time_t` is "now"

5.15.3.3 unsigned short mysqlpp::Date::year () const `[inline]`

Get the date's year part.

There's no trickery here like in some date implementations where you have to add 1900 or something like that.

Referenced by mysqlpp::DateTime::convert().

5.15.3.4 void mysqlpp::Date::year (unsigned short *y*) `[inline]`

Change the date's year part.

Pass the year value normally; we don't optimize the value by subtracting 1900 like some other date implementations.

The documentation for this class was generated from the following files:

- **datetime.h**
- datetime.cpp

5.16 mysqlpp::DateTime Class Reference

C++ form of SQL's DATETIME type.

`#include <datetime.h>`

Inheritance diagram for mysqlpp::DateTime:

Collaboration diagram for mysqlpp::DateTime:

Public Member Functions

- **DateTime** ()

  Default constructor.

- **DateTime** (unsigned short y, unsigned char mon, unsigned char d, unsigned char h, unsigned char min, unsigned char s)

  Initialize object from discrete y/m/d h:m:s values.

- **DateTime** (const **DateTime** &other)

  *Initialize object as a copy of another **Date** (p. 53).*

- **DateTime** (const char ∗str)

  Initialize object from a C string containing a SQL date-and-time string.

- template<class Str>
  DateTime (const Str &str)

  Initialize object from a C++ string containing a SQL date-and-time string.

- **DateTime** (time_t t)

  Initialize object from a `time_t`.

- int **compare** (const **DateTime** &other) const
  Compare this object to another.
- const char ∗ **convert** (const char ∗)
  Parse a SQL date and time string into this object.
- unsigned char **day** () const
  Get the date/time value's day part, 1-31.
- void **day** (unsigned char d)
  Change the date/time value's day part, 1-31.
- unsigned char **hour** () const
  Get the date/time value's hour part, 0-23.
- void **hour** (unsigned char h)
  Change the date/time value's hour part, 0-23.
- bool **is_now** () const
  Returns true if object will evaluate to SQL "NOW()" on conversion to string.
- unsigned char **minute** () const
  Get the date/time value's minute part, 0-59.
- void **minute** (unsigned char m)
  Change the date/time value's minute part, 0-59.
- unsigned char **month** () const
  Get the date/time value's month part, 1-12.
- void **month** (unsigned char m)
  Change the date/time value's month part, 1-12.
- **operator std::string** () const
  Convert to std::string.
- **operator time_t** () const
  Convert to time_t.
- unsigned char **second** () const
  Get the date/time value's second part, 0-59.

- void **second** (unsigned char s)

  Change the date/time value's second part, 0-59.

- std::string **str** () const

  Return our value in std::string form.

- unsigned short **year** () const

  Get the date/time value's year part.

- void **year** (unsigned short y)

  Change the date/time value's year part.

Static Public Member Functions

- static **DateTime now** ()

  Factory to create an object instance that will convert to SQL "NOW()" on insertion into a query.

5.16.1 Detailed Description

C++ form of SQL's DATETIME type.

This object exists primarily for conversion purposes. You can initialize it in several different ways, and then convert the object to SQL string form, extract the individual y/m/d h:m:s values, convert it to C's time_t, etc.

5.16.2 Constructor & Destructor Documentation

5.16.2.1 mysqlpp::DateTime::DateTime (unsigned short *y*, unsigned char *mon*, unsigned char *d*, unsigned char *h*, unsigned char *min*, unsigned char *s*) `[inline]`

Initialize object from discrete y/m/d h:m:s values.

Parameters:

y year_

mon month_

d day_ of month_

h hour_

min minute_

s second_

5.16.2.2 mysqlpp::DateTime::DateTime (const char ∗ *str*) `[inline, explicit]`

Initialize object from a C string containing a SQL date-and-time string.

String (p. 211) must be in the HH:MM:SS format. It doesn't have to be zero-padded.

5.16.2.3 template<class Str> mysqlpp::DateTime::DateTime (const Str & *str*) `[inline, explicit]`

Initialize object from a C++ string containing a SQL date-and-time string.

This works with any stringish class that declares a c_str() member function: std::string, **mysqlpp::String** (p. 211)...

See also:

DateTime(const char∗) (p. 60)

5.16.3 Member Function Documentation

5.16.3.1 int mysqlpp::DateTime::compare (const DateTime & *other*) const

Compare this object to another.

Returns < 0 if this object is before the other, 0 of they are equal, and > 0 if this object is after the other.

References now_.

5.16.3.2 static DateTime mysqlpp::DateTime::now () `[inline, static]`

Factory to create an object instance that will convert to SQL "NOW()" on insertion into a query.

This is just syntactic sugar around the default ctor

5.16.3.3 unsigned short mysqlpp::DateTime::year () const `[inline]`

Get the date/time value's year part.

There's no trickery here like in some date/time implementations where you have to add 1900 or something like that.

5.16.3.4 void mysqlpp::DateTime::year (unsigned short *y*) `[inline]`

Change the date/time value's year part.

Pass the year value normally; we don't optimize the value by subtracting 1900 like some other date/time implementations.

The documentation for this class was generated from the following files:

- **datetime.h**
- datetime.cpp

5.17 mysqlpp::DBDriver Class Reference

Provides a thin abstraction layer over the underlying database client library.

`#include <dbdriver.h>`

Public Types

- enum **nr_code** { **nr_more_results**, **nr_last_result**, **nr_error**, **nr_not_-supported** }

  Result code returned by ***next_result()*** *(*p. *73).*

Public Member Functions

- **DBDriver** ()

  Create object.

- **DBDriver** (const **DBDriver** &other)

  Duplicate an existing driver.

- virtual ~**DBDriver** ()

  Destroy object.

- ulonglong **affected_rows** ()

  Return the number of rows affected by the last query.

- std::string **client_version** () const

  Get database client library version.

- bool **connect** (const MYSQL &mysql)

  Establish a new connection using the same parameters as an existing connection.

- virtual bool **connect** (const char ∗host, const char ∗socket_name, unsigned int port, const char ∗db, const char ∗user, const char ∗password)

  Connect to database server.

- bool **connected** () const

  Return true if we have an active connection to the database server.

- void **copy** (const **DBDriver** &other)

  Establish a new connection as a copy of an existing one.

- bool **create_db** (const char ∗db) const

  Ask the database server to create a database.

- void **data_seek** (MYSQL_RES ∗res, ulonglong offset) const

  Seeks to a particualr row within the result set.

- void **disconnect** ()

  Drop the connection to the database server.

- bool **drop_db** (const std::string &db) const

  Drop a database.

- bool **enable_ssl** (const char ∗key=0, const char ∗cert=0, const char ∗ca=0, const char ∗capath=0, const char ∗cipher=0)

  Enable SSL-encrypted connection.

- const char ∗ **error** ()

  Return error message for last MySQL error associated with this connection.

- int **errnum** ()

  Return last MySQL error number associated with this connection.

- size_t **escape_string** (char ∗to, const char ∗from, size_t length)

  SQL-escapes the given string, taking into account the.

- bool **execute** (const char ∗qstr, size_t length)

  Executes the given query string.

- MYSQL_ROW **fetch_row** (MYSQL_RES ∗res) const

  Returns the next raw C API row structure from the given result set.

- const unsigned long ∗ **fetch_lengths** (MYSQL_RES ∗res) const

  Returns the lengths of the fields in the current row from a "use" query.

- MYSQL_FIELD ∗ **fetch_field** (MYSQL_RES ∗res, size_t i=UINT_MAX) const

  Returns information about a particular field in a result set.

- void **field_seek** (MYSQL_RES ∗res, size_t field) const

  Jumps to the given field within the result set.

- void **free_result** (MYSQL_RES ∗res) const

Releases memory used by a result set.

- st_mysql_options **get_options** () const

  Return the connection options object.

- std::string **ipc_info** ()

  Get information about the IPC connection to the database server.

- ulonglong **insert_id** ()

  Get ID generated for an AUTO_INCREMENT column in the previous INSERT query.

- bool **kill** (unsigned long tid)

  Kill a MySQL server thread.

- bool **more_results** ()

  Returns true if there are unconsumed results from the most recent query.

- **nr_code next_result** ()

  Moves to the next result set from a multi-query.

- int **num_fields** (MYSQL_RES ∗res) const

  Returns the number of fields in the given result set.

- ulonglong **num_rows** (MYSQL_RES ∗res) const

  Returns the number of rows in the given result set.

- bool **ping** ()

  "Pings" the MySQL database

- int **protocol_version** ()

  Returns version number of MySQL protocol this connection is using.

- std::string **query_info** ()

  Returns information about the last executed query.

- bool **refresh** (unsigned options)

  Asks the database server to refresh certain internal data structures.

- bool **result_empty** ()

  Returns true if the most recent result set was empty.

- bool **select_db** (const char ∗db)

Asks the database server to switch to a different database.

- std::string **server_version** ()

  Get the database server's version number.

- std::string **set_option** (**Option** ∗o)

  Sets a connection option.

- bool **set_option** (mysql_option moption, const void ∗arg=0)

  ***Set** (*p. *182) MySQL C API connection option.*

- bool **set_option** (unsigned int option, bool arg)

  ***Set** (*p. *182) MySQL C API connection option.*

- std::string **set_option_default** (**Option** ∗o)

  *Same as **set_option()** (*p. *74), except that it won't override a previously-set option.*

- bool **shutdown** ()

  Ask database server to shut down.

- std::string **server_status** ()

  Returns the database server's status.

- MYSQL_RES ∗ **store_result** ()

  *Saves the results of the query just **execute()** (*p. *70)d in memory and returns a pointer to the MySQL C API data structure the results are stored in.*

- unsigned long **thread_id** ()

  Returns the MySQL server thread ID for this connection.

- MYSQL_RES ∗ **use_result** ()

  Returns a result set from the last-executed query which we can walk through in linear fashion, which doesn't store all result sets in memory.

Static Public Member Functions

- static size_t **escape_string_no_conn** (char ∗to, const char ∗from, size_-t length)

  SQL-escapes the given string without reference to the.

- static bool **thread_aware** ()

Returns true if MySQL++ and the underlying MySQL C API library were both compiled with thread awareness.

- static void **thread_end** ()

  Tells the underlying MySQL C API library that this thread is done using the library.

- static bool **thread_start** ()

  Tells the underlying C API library that the current thread will be using the library's services.

5.17.1 Detailed Description

Provides a thin abstraction layer over the underlying database client library.

This class does as little as possible to adapt between its public interface and the interface required by the underlying C API. That is, in fact, its only mission. The high-level interfaces indended for use by MySQL++ users are in **Connection** (p. 32), **Query** (p. 126), Result, and ResUse, all of which delegate the actual database communication to an object of this type, created by **Connection** (p. 32). If you really need access to the low-level database driver, get it via **Connection::driver()** (p. 33); don't create **DBDriver** (p. 62) objects directly.

Currently this is a concrete class for wrapping the MySQL C API. In the future, it may be turned into an abstract base class, with subclasses for different database server types.

5.17.2 Member Enumeration Documentation

5.17.2.1 enum mysqlpp::DBDriver::nr_code

Result code returned by **next_result()** (p. 73).

Enumerator:

nr_more_results success, with more results to come

nr_last_result success, last result received

nr_error problem retrieving next result

nr_not_supported this C API doesn't support "next result"

5.17.3 Constructor & Destructor Documentation

5.17.3.1 mysqlpp::DBDriver::DBDriver (const DBDriver & *other*)

Duplicate an existing driver.

Parameters:

other existing **DBDriver** (p. 62) object

This establishes a new database server connection with the same parameters as the other driver's.

References copy().

5.17.4 Member Function Documentation

5.17.4.1 ulonglong mysqlpp::DBDriver::affected_rows () `[inline]`

Return the number of rows affected by the last query.

Wraps `mysql_affected_rows()` in the MySQL C API.

Referenced by mysqlpp::Query::affected_rows().

5.17.4.2 std::string mysqlpp::DBDriver::client_version () const `[inline]`

Get database client library version.

Wraps `mysql_get_client_info()` in the MySQL C API.

Referenced by mysqlpp::Connection::client_version(), and set_option().

5.17.4.3 bool mysqlpp::DBDriver::connect (const MYSQL & *mysql*)

Establish a new connection using the same parameters as an existing connection.

Parameters:

mysql existing MySQL C API connection object

References connected(), disconnect(), and set_option_default().

Referenced by mysqlpp::Connection::connect(), and copy().

5.17.4.4 bool mysqlpp::DBDriver::connect (const char ∗ *host*, const char ∗ *socket_name*, unsigned int *port*, const char ∗ *db*, const char ∗ *user*, const char ∗ *password*) `[virtual]`

Connect to database server.

If you call this method on an object that is already connected to a database server, the previous connection is dropped and a new connection is established.

References connected(), disconnect(), and set_option_default().

5.17.4.5 bool mysqlpp::DBDriver::connected () const `[inline]`

Return true if we have an active connection to the database server.

This does not actually check whether the connection is viable, it just indicates whether there was previously a successful **connect()** (p. 67) call and no **disconnect()** (p. 68). Call **ping()** (p. 73) to actually test the connection's viability.

Referenced by connect(), mysqlpp::Connection::connected(), copy(), and ∼DBDriver().

5.17.4.6 void mysqlpp::DBDriver::copy (const DBDriver & *other*)

Establish a new connection as a copy of an existing one.

Parameters:

other the connection to copy

References connect(), connected(), and mysql_.

Referenced by mysqlpp::Connection::copy(), and DBDriver().

5.17.4.7 bool mysqlpp::DBDriver::create_db (const char $*$ *db*) const

Ask the database server to create a database.

Parameters:

db name of database to create

Returns:

true if database was created successfully

5.17.4.8 void mysqlpp::DBDriver::data_seek (MYSQL_RES $*$ *res*, ulonglong *offset*) const `[inline]`

Seeks to a particualr row within the result set.

Wraps mysql_data_seek() in MySQL C API.

5.17.4.9 void mysqlpp::DBDriver::disconnect ()

Drop the connection to the database server.

This method is protected because it should only be used within the library. Unless you use the default constructor, this object should always be connected.

Referenced by connect(), mysqlpp::Connection::disconnect(), and ∼DBDriver().

5.17.4.10 bool mysqlpp::DBDriver::drop_db (const std::string & *db*) const

Drop a database.

Parameters:

db name of database to destroy

Returns:

true if database was created successfully

5.17.4.11 bool mysqlpp::DBDriver::enable_ssl (const char ∗ *key* = `0`, const char ∗ *cert* = `0`, const char ∗ *ca* = `0`, const char ∗ *capath* = `0`, const char ∗ *cipher* = `0`)

Enable SSL-encrypted connection.

Parameters:

key the pathname to the key file
cert the pathname to the certificate file
ca the pathname to the certificate authority file
capath directory that contains trusted SSL CA certificates in pem format.
cipher list of allowable ciphers to use

Returns:

False if call fails or the C API library wasn't compiled with SSL support enabled.

Must be called before connection is established.

Wraps `mysql_ssl_set()` in MySQL C API.

5.17.4.12 const char∗ mysqlpp::DBDriver::error () `[inline]`

Return error message for last MySQL error associated with this connection.

Can return a MySQL++ DBDriver-specific error message if there is one. If not, it simply wraps `mysql_error()` in the MySQL C API.

Referenced by mysqlpp::Connection::error().

5.17.4.13 int mysqlpp::DBDriver::errnum () `[inline]`

Return last MySQL error number associated with this connection.

Wraps `mysql_errno()` in the MySQL C API.

Referenced by mysqlpp::Connection::errnum().

5.17.4.14 size_t mysqlpp::DBDriver::escape_string (char ∗ *to*, const char ∗ *from*, size_t *length*) `[inline]`

SQL-escapes the given string, taking into account the.

default character set of the database server we're connected to. Wraps `mysql_-real_escape_string()` in the MySQL C API.

Referenced by mysqlpp::Query::escape_string().

5.17.4.15 static size_t mysqlpp::DBDriver::escape_string_no_conn (char ∗ *to*, const char ∗ *from*, size_t *length*) `[inline, static]`

SQL-escapes the given string without reference to the.

character set of a database server. Wraps `mysql_escape_string()` in the MySQL C API.

Referenced by mysqlpp::Query::escape_string().

5.17.4.16 bool mysqlpp::DBDriver::execute (const char ∗ *qstr*, size_t *length*) `[inline]`

Executes the given query string.

Wraps `mysql_real_query()` in the MySQL C API.

Referenced by mysqlpp::Query::exec(), mysqlpp::Query::execute(), mysqlpp::Query::store(), and mysqlpp::Query::use().

5.17.4.17 MYSQL_ROW mysqlpp::DBDriver::fetch_row (MYSQL_RES ∗ *res*) const `[inline]`

Returns the next raw C API row structure from the given result set.

This is for "use" query result sets only. "store" queries have all the rows already.

Wraps `mysql_fetch_row()` in MySQL C API.

Referenced by mysqlpp::UseQueryResult::fetch_raw_-row(), mysqlpp::UseQueryResult::fetch_row(), and mysqlpp::StoreQueryResult::StoreQueryResult().

5.17.4.18 const unsigned long∗ mysqlpp::DBDriver::fetch_lengths (MYSQL_RES ∗ *res*) const `[inline]`

Returns the lengths of the fields in the current row from a "use" query.

Wraps `mysql_fetch_lengths()` in MySQL C API.

Referenced by mysqlpp::UseQueryResult::fetch_lengths(), and mysqlpp::StoreQueryResult::StoreQueryResult().

5.17.4.19 MYSQL_FIELD∗ mysqlpp::DBDriver::fetch_field (MYSQL_RES ∗ *res*, size_t *i* = `UINT_MAX`) const `[inline]`

Returns information about a particular field in a result set.

Parameters:

res result set to fetch field information for

i field number to fetch information for, if given

If i parameter is given, this call is like a combination of **field_seek()** (p. 71) followed by **fetch_field()** (p. 71) without the i parameter, which otherwise just iterates through the set of fields in the given result set.

Wraps `mysql_fetch_field()` and mysql_fetch_field_direct() in MySQL C API. (Which one it uses depends on i parameter.)

Referenced by mysqlpp::ResultBase::ResultBase().

5.17.4.20 void mysqlpp::DBDriver::field_seek (MYSQL_RES ∗ *res*, size_t *field*) const `[inline]`

Jumps to the given field within the result set.

Wraps `mysql_field_seek()` in MySQL C API.

Referenced by mysqlpp::ResultBase::ResultBase().

5.17.4.21 void mysqlpp::DBDriver::free_result (MYSQL_RES ∗ *res*) const `[inline]`

Releases memory used by a result set.

Wraps `mysql_free_result()` in MySQL C API.

Referenced by mysqlpp::StoreQueryResult::StoreQueryResult().

5.17.4.22 std::string mysqlpp::DBDriver::ipc_info () `[inline]`

Get information about the IPC connection to the database server.

String (p. 211) contains info about type of connection (e.g. TCP/IP, named pipe, Unix socket...) and the server hostname.

Wraps `mysql_get_host_info()` in the MySQL C API.

Referenced by mysqlpp::Connection::ipc_info().

5.17.4.23 ulonglong mysqlpp::DBDriver::insert_id () `[inline]`

Get ID generated for an AUTO_INCREMENT column in the previous INSERT query.

Return values:

0 if the previous query did not generate an ID. Use the SQL function LAST_-INSERT_ID() if you need the last ID generated by any query, not just the previous one.

Referenced by mysqlpp::Query::insert_id().

5.17.4.24 bool mysqlpp::DBDriver::kill (unsigned long *tid*) `[inline]`

Kill a MySQL server thread.

Parameters:

tid ID of thread to kill

Wraps `mysql_kill()` in the MySQL C API.

See also:

thread_id() (p. 76)

Referenced by mysqlpp::Connection::kill().

5.17.4.25 bool mysqlpp::DBDriver::more_results () `[inline]`

Returns true if there are unconsumed results from the most recent query.

Wraps `mysql_more_results()` in the MySQL C API.

Referenced by mysqlpp::Query::more_results().

5.17.4.26 nr_code mysqlpp::DBDriver::next_result () `[inline]`

Moves to the next result set from a multi-query.

Returns:

A code indicating whether we successfully found another result, there were no more results (but still success) or encountered an error trying to find the next result set.

Wraps `mysql_next_result()` in the MySQL C API, with translation of its return value from magic integers to nr_code enum values.

References nr_error, nr_last_result, nr_more_results, and nr_not_supported.

Referenced by mysqlpp::Query::store_next().

5.17.4.27 int mysqlpp::DBDriver::num_fields (MYSQL_RES ∗ *res*) const `[inline]`

Returns the number of fields in the given result set.

Wraps `mysql_num_fields()` in MySQL C API.

5.17.4.28 ulonglong mysqlpp::DBDriver::num_rows (MYSQL_RES ∗ *res*) const `[inline]`

Returns the number of rows in the given result set.

Wraps `mysql_num_rows()` in MySQL C API.

5.17.4.29 bool mysqlpp::DBDriver::ping () `[inline]`

"Pings" the MySQL database

This function will try to reconnect to the server if the connection has been dropped. Wraps `mysql_ping()` in the MySQL C API.

Return values:

true if server is responding, regardless of whether we had to reconnect or not

false if either we already know the connection is down and cannot re-establish it, or if the server did not respond to the ping and we could not re-establish the connection.

Referenced by mysqlpp::Connection::ping().

5.17.4.30 int mysqlpp::DBDriver::protocol_version () `[inline]`

Returns version number of MySQL protocol this connection is using.

Wraps `mysql_get_proto_info()` in the MySQL C API.

Referenced by mysqlpp::Connection::protocol_version().

5.17.4.31 string mysqlpp::DBDriver::query_info ()

Returns information about the last executed query.

Wraps `mysql_info()` in the MySQL C API

Referenced by mysqlpp::Query::info().

5.17.4.32 bool mysqlpp::DBDriver::refresh (unsigned *options*) `[inline]`

Asks the database server to refresh certain internal data structures.

Wraps `mysql_refresh()` in the MySQL C API. There is no corresponding interface for this in higher level MySQL++ classes because it was undocumented until recently, and it's a pretty low-level thing. It's designed for things like MySQL Administrator.

5.17.4.33 bool mysqlpp::DBDriver::result_empty () `[inline]`

Returns true if the most recent result set was empty.

Wraps `mysql_field_count()` in the MySQL C API, returning true if it returns 0.

5.17.4.34 std::string mysqlpp::DBDriver::server_version () `[inline]`

Get the database server's version number.

Wraps `mysql_get_server_info()` in the MySQL C API.

Referenced by mysqlpp::Connection::server_version().

5.17.4.35 std::string mysqlpp::DBDriver::set_option (Option $*$ *o*)

Sets a connection option.

This is the database-independent high-level option setting interface that **Connection::set_option()** (p. 40) calls. There are several private overloads that actually implement the option setting.

See also:

Connection::set_option(Option∗) (p. 40) for commentary

References client_version(), mysqlpp::Option::err_api_limit, mysqlpp::Option::err_-api_reject, mysqlpp::Option::err_connected, mysqlpp::Option::err_NONE, and mysqlpp::Option::set().

Referenced by mysqlpp::Connection::set_option(), and set_option_default().

5.17.4.36 bool mysqlpp::DBDriver::set_option (unsigned int *option*, bool *arg*)

Set (p. 182) MySQL C API connection option.

Manipulates the MYSQL.client_flag bit mask. This allows these flags to be treated the same way as any other connection option, even though the C API handles them differently.

5.17.4.37 bool mysqlpp::DBDriver::shutdown ()

Ask database server to shut down.

User must have the "shutdown" privilege.

Wraps `mysql_shutdown()` in the MySQL C API.

Referenced by mysqlpp::Connection::shutdown().

5.17.4.38 std::string mysqlpp::DBDriver::server_status () `[inline]`

Returns the database server's status.

String (p. 211) is similar to that returned by the `mysqladmin status` command. Among other things, it contains uptime in seconds, and the number of running threads, questions and open tables.

Wraps `mysql_stat()` in the MySQL C API.

Referenced by mysqlpp::Connection::server_status().

5.17.4.39 MYSQL_RES∗ mysqlpp::DBDriver::store_result () `[inline]`

Saves the results of the query just **execute()** (p. 70)d in memory and returns a pointer to the MySQL C API data structure the results are stored in.

See also:

use_result() (p. 77)

Wraps `mysql_store_result()` in the MySQL C API.

Referenced by mysqlpp::Query::store(), and mysqlpp::Query::store_next().

5.17.4.40 bool mysqlpp::DBDriver::thread_aware () `[static]`

Returns true if MySQL++ and the underlying MySQL C API library were both compiled with thread awareness.

This is based in part on a MySQL C API function mysql_thread_safe(). We deliberately don't call this wrapper thread_safe() because it's a misleading name: linking to thread-aware versions of the MySQL++ and C API libraries doesn't automatically make your program "thread-safe". See the `chapter on threads` in the user manual for more information and guidance.

Referenced by mysqlpp::Connection::thread_aware().

5.17.4.41 static void mysqlpp::DBDriver::thread_end () `[inline, static]`

Tells the underlying MySQL C API library that this thread is done using the library.

This exists because the MySQL C API library allocates some per-thread memory which it doesn't release until you call this.

Referenced by mysqlpp::Connection::thread_end().

5.17.4.42 unsigned long mysqlpp::DBDriver::thread_id () `[inline]`

Returns the MySQL server thread ID for this connection.

This has nothing to do with threading on the client side. It's a server-side thread ID, to be used with **kill()** (p. 72).

Referenced by mysqlpp::Connection::thread_id().

5.17.4.43 static bool mysqlpp::DBDriver::thread_start () `[inline, static]`

Tells the underlying C API library that the current thread will be using the library's services.

Return values:

True if there was no problem

The MySQL++ user manual's `chapter on threads` details two major strategies for dealing with connections in the face of threads. If you take the simpler path, creating one **DBDriver** (p. 62) object per thread, it is never necessary to call this function; the underlying C API will call it for you when you establish the first database server connection from that thread. If you use a more complex connection management strategy where it's possible for one thread to establish a connection that another thread uses, you must call this from each thread that can use the database before it creates any MySQL++ objects. If you use a DBDriverPool object, this applies; DBDriverPool isn't smart enough to call this for you, and the MySQL C API won't do it, either.

Referenced by mysqlpp::Connection::thread_start().

5.17.4.44 MYSQL_RES∗ mysqlpp::DBDriver::use_result () `[inline]`

Returns a result set from the last-executed query which we can walk through in linear fashion, which doesn't store all result sets in memory.

See also:

store_result (p. 75)

Wraps `mysql_use_result()` in the MySQL C API.

Referenced by mysqlpp::Query::use().

The documentation for this class was generated from the following files:

- **dbdriver.h**
- dbdriver.cpp

5.18 mysqlpp::DBSelectionFailed Class Reference

Exception (p. 84) thrown when the program tries to select a new database and the database server refuses for some reason.

`#include <exceptions.h>`

Inheritance diagram for mysqlpp::DBSelectionFailed:

Collaboration diagram for mysqlpp::DBSelectionFailed:

Public Member Functions

- **DBSelectionFailed** (const char ∗w="", int e=0)

  Create exception object.

- int **errnum** () const

  *Return the error number corresponding to the error message returned by **what()** (p. 85), if any.*

5.18.1 Detailed Description

Exception (p. 84) thrown when the program tries to select a new database and the database server refuses for some reason.

5.18.2 Constructor & Destructor Documentation

5.18.2.1 mysqlpp::DBSelectionFailed::DBSelectionFailed (const char * *w* = "", int *e* = 0) `[inline, explicit]`

Create exception object.

Parameters:

- ***w*** explanation for why the exception was thrown
- ***e*** the error number from the underlying database API

5.18.3 Member Function Documentation

5.18.3.1 int mysqlpp::DBSelectionFailed::errnum () const `[inline]`

Return the error number corresponding to the error message returned by **what()** (p. 85), if any.

If the error number is 0, it means that the error message doesn't come from the underlying database API, but rather from MySQL++ itself. This happens when an error condition is detected up at this higher level instead of letting the underlying database API do it.

The documentation for this class was generated from the following file:

- **exceptions.h**

5.19 mysqlpp::equal_list_b< Seq1, Seq2, Manip > Struct Template Reference

Same as **equal_list_ba** (p. 82), plus the option to have some elements of the equals clause suppressed.

`#include <vallist.h>`

Collaboration diagram for mysqlpp::equal_list_b< Seq1, Seq2, Manip >:

Public Member Functions

- **equal_list_b** (const Seq1 &s1, const Seq2 &s2, const std::vector< bool > &f, const char ∗d, const char ∗e, Manip m)

  Create object.

Public Attributes

- const Seq1 ∗ **list1**

  the list of objects on the left-hand side of the equals sign

- const Seq2 ∗ **list2**

  the list of objects on the right-hand side of the equals sign

- const std::vector< bool > **fields**

  for each true item in the list, the pair in that position will be inserted into a C++ stream

- const char ∗ **delim**

  delimiter to use between each pair of elements

- const char ∗ **equl**

  "equal" sign to use between each item in each equal pair; doesn't have to actually be " = "

- Manip **manip**

  manipulator to use when inserting the equal_list into a C++ stream

5.19.1 Detailed Description

template<class Seq1, class Seq2, class Manip> struct mysqlpp::equal_list_b< Seq1, Seq2, Manip >

Same as **equal_list_ba** (p. 82), plus the option to have some elements of the equals clause suppressed.

Imagine an object of this type contains the lists (a, b, c) (d, e, f), that the object's 'fields' list is (true, false, true), and that the object's delimiter and equals symbols are set to " AND " and " = " respectively. When you insert that object into a C++ stream, you would get "a = d AND c = f".

See equal_list_ba's documentation for more details.

5.19.2 Constructor & Destructor Documentation

5.19.2.1 template<class Seq1, class Seq2, class Manip> mysqlpp::equal_list_b< Seq1, Seq2, Manip >::equal_list_b (const Seq1 & *s1*, const Seq2 & *s2*, const std::vector< bool > & *f*, const char ∗ *d*, const char ∗ *e*, Manip *m*) `[inline]`

Create object.

Parameters:

s1 list of objects on left-hand side of equal sign

s2 list of objects on right-hand side of equal sign

f for each true item in the list, the pair of items in that position will be inserted into a C++ stream

d what delimiter to use between each group in the list when inserting the list into a C++ stream

e the "equals" sign between each pair of items in the equal list; doesn't actually have to be " = "!

m manipulator to use when inserting the list into a C++ stream

The documentation for this struct was generated from the following file:

- **vallist.h**

5.20 mysqlpp::equal_list_ba< Seq1, Seq2, Manip > Struct Template Reference

Holds two lists of items, typically used to construct a SQL "equals clause".

`#include <vallist.h>`

Collaboration diagram for mysqlpp::equal_list_ba< Seq1, Seq2, Manip >:

Public Member Functions

- **equal_list_ba** (const Seq1 &s1, const Seq2 &s2, const char ∗d, const char ∗e, Manip m)

  Create object.

Public Attributes

- const Seq1 ∗ **list1**

  the list of objects on the left-hand side of the equals sign

- const Seq2 ∗ **list2**

  the list of objects on the right-hand side of the equals sign

- const char ∗ **delim**

  delimiter to use between each pair of elements

- const char ∗ **equl**

  "equal" sign to use between each item in each equal pair; doesn't have to actually be " = "

- Manip **manip**

  manipulator to use when inserting the equal_list into a C++ stream

5.20.1 Detailed Description

template<class Seq1, class Seq2, class Manip> struct mysqlpp::equal_list_ba< Seq1, Seq2, Manip >

Holds two lists of items, typically used to construct a SQL "equals clause".

The WHERE clause in a SQL SELECT statment is an example of an equals clause.

Imagine an object of this type contains the lists (a, b) (c, d), and that the object's delimiter and equals symbols are set to ", " and " = " respectively. When you insert that object into a C++ stream, you would get "a = c, b = d".

This class is never instantiated by hand. The equal_list() functions build instances of this structure template to do their work. MySQL++'s SSQLS mechanism calls those functions when building SQL queries; you can call them yourself to do similar work. The "Harnessing SSQLS Internals" section of the user manual has some examples of this.

See also:

equal_list_b (p. 80)

5.20.2 Constructor & Destructor Documentation

5.20.2.1 template<class Seq1, class Seq2, class Manip> mysqlpp::equal_list_ba< Seq1, Seq2, Manip >::equal_list_ba (const Seq1 & *s1*, const Seq2 & *s2*, const char * *d*, const char * *e*, Manip *m*) `[inline]`

Create object.

Parameters:

s1 list of objects on left-hand side of equal sign

s2 list of objects on right-hand side of equal sign

d what delimiter to use between each group in the list when inserting the list into a C++ stream

e the "equals" sign between each pair of items in the equal list; doesn't actually have to be " = "!

m manipulator to use when inserting the list into a C++ stream

The documentation for this struct was generated from the following file:

- **vallist.h**

5.21 mysqlpp::Exception Class Reference

Base class for all MySQL++ custom exceptions.

`#include <exceptions.h>`

Inheritance diagram for mysqlpp::Exception:

Collaboration diagram for mysqlpp::Exception:

Public Member Functions

- **Exception** (const **Exception** &e) throw ()

  Create exception object as copy of another.

- **Exception** & **operator=** (const **Exception** &rhs) throw ()

  Assign another exception object's contents to this one.

- ∼**Exception** () throw ()

  Destroy exception object.

- virtual const char ∗ **what** () const throw ()

  Returns explanation of why exception was thrown.

Protected Member Functions

- **Exception** (const char ∗w="") throw ()

  Create exception object.

- **Exception** (const std::string &w) throw ()

  Create exception object.

Protected Attributes

- std::string **what_**

  explanation of why exception was thrown

5.21.1 Detailed Description

Base class for all MySQL++ custom exceptions.

The documentation for this class was generated from the following file:

- **exceptions.h**

5.22 mysqlpp::Field Class Reference

Class to hold information about a SQL field.

`#include <field.h>`

Collaboration diagram for mysqlpp::Field:

Public Member Functions

- **Field** ()

  Create empty object.

- **Field** (const MYSQL_FIELD ∗pf)

  Create object from C API field structure.

- **Field** (const **Field** &other)

  *Create object as a copy of another **Field** (p. 87).*

- bool **auto_increment** () const

  Returns true if field auto-increments.

- bool **binary_type** () const

  Returns true if field is of some binary type.

- bool **blob_type** () const

  Returns true if field is of some BLOB type.

- const char ∗ **db** () const

  Return the name of the database the field comes from.

- bool **enumeration** () const

  Returns true if field is of an enumerated value type.

- size_t **length** () const

Return the creation size of the field.

- size_t **max_length** () const

  Return the maximum number of bytes stored in this field in any of the rows in the result set we were created from.

- bool **multiple_key** () const

  Returns true if field is part of a key.

- const char ∗ **name** () const

  Return the field's name.

- bool **primary_key** () const

  Returns true if field is part of a primary key.

- bool **set_type** () const

  Returns true if field is of some 'set' type.

- const char ∗ **table** () const

  Return the name of the table the field comes from.

- bool **timestamp** () const

  Returns true if field's type is timestamp.

- const **mysql_type_info** & **type** () const

  Return information about the field's type.

- bool **unique_key** () const

  Returns true if field is part of a unique key.

- bool **zerofill** () const

  Returns true if field has the zerofill attribute.

5.22.1 Detailed Description

Class to hold information about a SQL field.

This is a cut-down version of MYSQL_FIELD, using MySQL++ and generic C++ types instead of the C types it uses, and hiding all fields behind accessors. It leaves out data members we have decided aren't very useful. Given a good argument, we're willing to mirror more of the fields; we just don't want to mirror the underlying structure slavishly for no benefit.

5.22.2 Member Function Documentation

5.22.2.1 size_t mysqlpp::Field::length () const `[inline]`

Return the creation size of the field.

This is the number of bytes the field can hold, not how much is actually stored in the field on any particular row.

The documentation for this class was generated from the following file:

- **field.h**

5.23 mysqlpp::FieldNames Class Reference

Holds a list of SQL field names.

`#include <field_names.h>`

Public Member Functions

- **FieldNames** ()
  Default constructor.
- **FieldNames** (const **FieldNames** &other)
  Copy constructor.
- **FieldNames** (const **ResultBase** ∗res)
  Create field name list from a result set.
- **FieldNames** (int i)
  Create empty field name list, reserving space for a fixed number of field names.
- **FieldNames** & **operator=** (const **ResultBase** ∗res)
  Initializes the field list from a result set.
- **FieldNames** & **operator=** (int i)
  Insert `i` empty field names at beginning of list.
- std::string & **operator[ ]** (int i)
  Get the name of a field given its index.
- const std::string & **operator[ ]** (int i) const
  Get the name of a field given its index, in const context.
- unsigned int **operator[ ]** (const std::string &s) const
  Get the index number of a field given its name.

5.23.1 Detailed Description

Holds a list of SQL field names.

The documentation for this class was generated from the following files:

- **field_names.h**
- field_names.cpp

5.24 mysqlpp::FieldTypes Class Reference

A vector of SQL field types.

`#include <field_types.h>`

Public Member Functions

- **FieldTypes** ()

  Default constructor.

- **FieldTypes** (const **ResultBase** ∗res)

  Create list of field types from a result set.

- **FieldTypes** (int i)

  Create fixed-size list of uninitialized field types.

- **FieldTypes** & **operator=** (const **ResultBase** ∗res)

  Initialize field list based on a result set.

- **FieldTypes** & **operator=** (int i)

  Insert a given number of uninitialized field type objects at the beginning of the list.

5.24.1 Detailed Description

A vector of SQL field types.

5.24.2 Member Function Documentation

5.24.2.1 FieldTypes& mysqlpp::FieldTypes::operator= (int *i*) `[inline]`

Insert a given number of uninitialized field type objects at the beginning of the list.

Parameters:

i number of field type objects to insert

The documentation for this class was generated from the following files:

- **field_types.h**
- field_types.cpp

5.25 mysqlpp::FoundRowsOption Class Reference

Make **Query::affected_rows()** (p. 126) return number of matched rows.

`#include <options.h>`

Inheritance diagram for mysqlpp::FoundRowsOption:

Collaboration diagram for mysqlpp::FoundRowsOption:

5.25.1 Detailed Description

Make **Query::affected_rows()** (p. 126) return number of matched rows.

Default is to return number of **changed** rows.

The documentation for this class was generated from the following file:

- **options.h**

5.26 mysqlpp::GuessConnectionOption Class Reference

Allow C API to guess what kind of connection to use.

`#include <options.h>`

Inheritance diagram for mysqlpp::GuessConnectionOption:

Collaboration diagram for mysqlpp::GuessConnectionOption:

5.26.1 Detailed Description

Allow C API to guess what kind of connection to use.

This is the default. The option exists to override **UseEmbeddedConnectionOption** (p. 245) and **UseEmbeddedConnectionOption** (p. 245).

The documentation for this class was generated from the following file:

- **options.h**

5.27 mysqlpp::IgnoreSpaceOption Class Reference

Allow spaces after function names in queries.

`#include <options.h>`

Inheritance diagram for mysqlpp::IgnoreSpaceOption:

Collaboration diagram for mysqlpp::IgnoreSpaceOption:

5.27.1 Detailed Description

Allow spaces after function names in queries.

The documentation for this class was generated from the following file:

- **options.h**

5.28 mysqlpp::InitCommandOption Class Reference

Give SQL executed on connect.

`#include <options.h>`

Inheritance diagram for mysqlpp::InitCommandOption:

Collaboration diagram for mysqlpp::InitCommandOption:

5.28.1 Detailed Description

Give SQL executed on connect.

The documentation for this class was generated from the following file:

- **options.h**

5.29 mysqlpp::InteractiveOption Class Reference

Assert that this is an interactive program.

`#include <options.h>`

Inheritance diagram for mysqlpp::InteractiveOption:

Collaboration diagram for mysqlpp::InteractiveOption:

5.29.1 Detailed Description

Assert that this is an interactive program.

Affects connection timeouts.

The documentation for this class was generated from the following file:

- **options.h**

5.30 mysqlpp::LocalFilesOption Class Reference

Enable LOAD DATA LOCAL statement.

```
#include <options.h>
```

Inheritance diagram for mysqlpp::LocalFilesOption:

Collaboration diagram for mysqlpp::LocalFilesOption:

5.30.1 Detailed Description

Enable LOAD DATA LOCAL statement.

The documentation for this class was generated from the following file:

- **options.h**

5.31 mysqlpp::LocalInfileOption Class Reference

Enable LOAD LOCAL INFILE statement.

`#include <options.h>`

Inheritance diagram for mysqlpp::LocalInfileOption:

Collaboration diagram for mysqlpp::LocalInfileOption:

5.31.1 Detailed Description

Enable LOAD LOCAL INFILE statement.

The documentation for this class was generated from the following file:

- **options.h**

5.32 mysqlpp::MultiResultsOption Class Reference

Enable multiple result sets in a reply.

`#include <options.h>`

Inheritance diagram for mysqlpp::MultiResultsOption:

Collaboration diagram for mysqlpp::MultiResultsOption:

5.32.1 Detailed Description

Enable multiple result sets in a reply.

The documentation for this class was generated from the following file:

- **options.h**

5.33 mysqlpp::MultiStatementsOption Class Reference

Enable multiple queries in a request to the server.

`#include <options.h>`

Inheritance diagram for mysqlpp::MultiStatementsOption:

Collaboration diagram for mysqlpp::MultiStatementsOption:

5.33.1 Detailed Description

Enable multiple queries in a request to the server.

The documentation for this class was generated from the following file:

- **options.h**

5.34 mysqlpp::MutexFailed Class Reference

Exception (p. 84) thrown when a **BeecryptMutex** (p. 26) object fails.

```
#include <exceptions.h>
```

Inheritance diagram for mysqlpp::MutexFailed:

Collaboration diagram for mysqlpp::MutexFailed:

Public Member Functions

- **MutexFailed** (const char ∗w="lock failed")
  Create exception object.

5.34.1 Detailed Description

Exception (p. 84) thrown when a **BeecryptMutex** (p. 26) object fails.

The documentation for this class was generated from the following file:

- **exceptions.h**

5.35 mysqlpp::mysql_type_info Class Reference

SQL field type information.

`#include <type_info.h>`

Public Member Functions

- **mysql_type_info** ()

  Default constructor.

- **mysql_type_info** (enum_field_types t, bool _unsigned=false, bool _-null=false)

  Create object from MySQL C API type info.

- **mysql_type_info** (const **mysql_type_info** &t)

  Create object as a copy of another.

- **mysql_type_info** (const std::type_info &t)

  Create object from a C++ type_info object.

- **mysql_type_info** & **operator=** (const **mysql_type_info** &t)

  *Assign another **mysql_type_info** (p. 102) object to this object.*

- **mysql_type_info** & **operator=** (const std::type_info &t)

  Assign a C++ type_info object to this object.

- const char ∗ **name** () const

  Returns an implementation-defined name of the C++ type.

- const char ∗ **sql_name** () const

  Returns the name of the SQL type.

- const std::type_info & **c_type** () const

  Returns the type_info for the C++ type associated with the SQL type.

- const **mysql_type_info base_type** () const

  *Returns the type_info for the C++ type inside of the **mysqlpp::Null** (p. 111) type.*

- int **id** () const

  Returns the ID of the SQL type.

- bool **quote_q** () const

Returns true if the SQL type is of a type that needs to be quoted.

- bool **escape_q** () const

  Returns true if the SQL type is of a type that needs to be escaped.

- bool **before** (**mysql_type_info** &b)

  Provides a way to compare two types for sorting.

Static Public Attributes

- static const enum_field_types **string_type**

  The internal constant we use for our string type.

5.35.1 Detailed Description

SQL field type information.

5.35.2 Constructor & Destructor Documentation

5.35.2.1 mysqlpp::mysql_type_info::mysql_type_info () `[inline]`

Default constructor.

This only exists because **FieldTypes** (p. 91) keeps a vector of these objects. You are expected to copy real values into it before using it via the copy ctor or one of the assignment operators. If you don't, we have arranged a pretty spectacular crash for your program. So there.

5.35.2.2 mysqlpp::mysql_type_info::mysql_type_info (enum_field_types *t*, bool *_unsigned* = `false`, bool *_null* = `false`) `[inline]`

Create object from MySQL C API type info.

Parameters:

t the underlying C API type ID for this type

_unsigned if true, this is the unsigned version of the type

_null if true, this type can hold a SQL null

5.35.2.3 mysqlpp::mysql_type_info::mysql_type_info (const std::type_info & *t*) `[inline]`

Create object from a C++ type_info object.

This tries to map a C++ type to the closest MySQL data type. It is necessarily somewhat approximate.

5.35.3 Member Function Documentation

5.35.3.1 mysql_type_info& mysqlpp::mysql_type_info::operator= (const std::type_info & *t*) `[inline]`

Assign a C++ type_info object to this object.

This tries to map a C++ type to the closest MySQL data type. It is necessarily somewhat approximate.

5.35.3.2 const char* mysqlpp::mysql_type_info::name () const `[inline]`

Returns an implementation-defined name of the C++ type.

Returns the name that would be returned by typeid().**name**() (p. 104) for the C++ type associated with the SQL type.

5.35.3.3 const char* mysqlpp::mysql_type_info::sql_name () const `[inline]`

Returns the name of the SQL type.

Returns the SQL name for the type.

5.35.3.4 const std::type_info& mysqlpp::mysql_type_info::c_type () const `[inline]`

Returns the type_info for the C++ type associated with the SQL type.

Returns the C++ type_info record corresponding to the SQL type.

Referenced by escape_q(), quote_q(), and mysqlpp::SQLBuffer::quote_q().

5.35.3.5 const mysql_type_info mysqlpp::mysql_type_info::base_type () const `[inline]`

Returns the type_info for the C++ type inside of the **mysqlpp::Null** (p. 111) type.

Returns the type_info for the C++ type inside the **mysqlpp::Null** (p. 111) type. If the type is not **Null** (p. 111) then this is the same as **c_type()** (p. 104).

Referenced by quote_q(), and mysqlpp::SQLBuffer::quote_q().

5.35.3.6 int mysqlpp::mysql_type_info::id () const `[inline]`

Returns the ID of the SQL type.

Returns the ID number MySQL uses for this type. Note: Do not depend on the value of this ID as it may change between MySQL versions.

5.35.3.7 bool mysqlpp::mysql_type_info::quote_q () const

Returns true if the SQL type is of a type that needs to be quoted.

Returns:

true if the type needs to be quoted for syntactically correct SQL.

References base_type(), and c_type().

Referenced by mysqlpp::SQLBuffer::quote_q().

5.35.3.8 bool mysqlpp::mysql_type_info::escape_q () const

Returns true if the SQL type is of a type that needs to be escaped.

Returns:

true if the type needs to be escaped for syntactically correct SQL.

References c_type().

Referenced by mysqlpp::SQLBuffer::escape_q().

5.35.3.9 bool mysqlpp::mysql_type_info::before (mysql_type_info & *b*) `[inline]`

Provides a way to compare two types for sorting.

Returns true if the SQL ID of this type is lower than that of another. Used by mysqlpp::type_info_cmp when comparing types.

References num_.

5.35.4 Member Data Documentation

5.35.4.1 const enum_field_types mysqlpp::mysql_type_info::string_type `[static]`

Initial value:

```
FIELD_TYPE_STRING
```

The internal constant we use for our string type.

We expose this because other parts of MySQL++ need to know what the string constant is at the moment.

Referenced by mysqlpp::SQLTypeAdapter::assign(), mysqlpp::SQLBuffer::is_-string(), mysqlpp::String::it_is_null(), mysqlpp::String::operator=(), and mysqlpp::String::type().

The documentation for this class was generated from the following files:

- **type_info.h**
- type_info.cpp

5.36 mysqlpp::NamedPipeOption Class Reference

Suggest use of named pipes.

`#include <options.h>`

Inheritance diagram for mysqlpp::NamedPipeOption:

Collaboration diagram for mysqlpp::NamedPipeOption:

5.36.1 Detailed Description

Suggest use of named pipes.

The documentation for this class was generated from the following file:

- **options.h**

5.37 mysqlpp::NoExceptions Class Reference

Disable exceptions in an object derived from **OptionalExceptions** (p. 123).

`#include <noexceptions.h>`

Collaboration diagram for mysqlpp::NoExceptions:

Public Member Functions

- **NoExceptions** (const **OptionalExceptions** &a)

  Constructor.

- ~**NoExceptions** ()

  Destructor.

5.37.1 Detailed Description

Disable exceptions in an object derived from **OptionalExceptions** (p. 123).

This class was designed to be created on the stack, taking a reference to a subclass of **OptionalExceptions** (p. 123). (We call that our "associate" object.) On creation, we save that object's current exception state, and disable exceptions. On destruction, we restore our associate's previous state.

5.37.2 Constructor & Destructor Documentation

5.37.2.1 mysqlpp::NoExceptions::NoExceptions (const OptionalExceptions & *a*) `[inline]`

Constructor.

Takes a reference to an **OptionalExceptions** (p. 123) derivative, saves that object's current exception state, and disables exceptions.

5.37.2.2 mysqlpp::NoExceptions::∼NoExceptions () `[inline]`

Destructor.

Restores our associate object's previous exception state.

The documentation for this class was generated from the following file:

- **noexceptions.h**

5.38 mysqlpp::NoSchemaOption Class Reference

Disable db.tbl.col syntax in queries.

`#include <options.h>`

Inheritance diagram for mysqlpp::NoSchemaOption:

Collaboration diagram for mysqlpp::NoSchemaOption:

5.38.1 Detailed Description

Disable db.tbl.col syntax in queries.

The documentation for this class was generated from the following file:

- **options.h**

5.39 mysqlpp::Null< Type, Behavior > Class Template Reference

Class for holding data from a SQL column with the NULL attribute.

`#include <null.h>`

Collaboration diagram for mysqlpp::Null< Type, Behavior >:

Public Types

- typedef Type **value_type**

    Type of the data stored in this object, when it is not equal to SQL null.

Public Member Functions

- **Null** ()

    Default constructor.

- **Null** (const Type &x)

    Initialize the object with a particular value.

- **Null** (const **null_type** &)

    *Construct a **Null** (p. 111) equal to SQL null.*

- **operator Type &** ()

    Converts this object to Type.

- **Null** & **operator=** (const Type &x)

    Assign a value to the object.

- **Null** & **operator=** (const **null_type** &n)

    Assign SQL null to this object.

- bool **operator==** (const **Null**< Type > &rhs) const

Do equality comparison of two nullable values.

- bool **operator==** (const **null_type** &) const

  Do equality comparison against hard-coded SQL null.

- bool **operator!=** (const **Null**< Type > &rhs) const

  Do inequality comparison of two nullable values.

- bool **operator!=** (const **null_type** &rhs) const

  Do inequality comparison against hard-coded SQL null.

- bool **operator**< (const **Null**< Type > &rhs) const

  Do less-than comparison of two nullable values.

- bool **operator**< (const **null_type** &) const

  Do less-than comparison against hard-coded SQL null.

Public Attributes

- Type **data**

  The object's value, when it is not SQL null.

- bool **is_null**

  If set, this object is considered equal to SQL null.

5.39.1 Detailed Description

template<class Type, class Behavior = NullIsNull> class mysqlpp::Null< Type, Behavior >

Class for holding data from a SQL column with the NULL attribute.

This template is necessary because there is nothing in the C++ type system with the same semantics as SQL's null. In SQL, a column can have the optional 'NULL' attribute, so there is a difference in type between, say an `int` column that can be null and one that cannot be. C++'s NULL constant does not have these features.

It's important to realize that this class doesn't hold nulls, it holds data that *can be* null. It can hold a non-null value, you can then assign null to it (using MySQL++'s global `null` object), and then assign a regular value to it again; the object will behave as you expect throughout this process.

Because one of the template parameters is a C++ type, the typeid() for a null `int` is different than for a null `string`, to pick two random examples. See type_info.cpp for the table SQL types that can be null.

5.39.2 Constructor & Destructor Documentation

5.39.2.1 template<class Type, class Behavior = NullIsNull> mysqlpp::Null< Type, Behavior >::Null () `[inline]`

Default constructor.

"data" member is left uninitialized by this ctor, because we don't know what to initialize it to.

5.39.2.2 template<class Type, class Behavior = NullIsNull> mysqlpp::Null< Type, Behavior >::Null (const Type & *x*) `[inline]`

Initialize the object with a particular value.

The object is marked as "not null" if you use this ctor. This behavior exists because the class doesn't encode nulls, but rather data which *can be* null. The distinction is necessary because 'NULL' is an optional attribute of SQL columns.

5.39.2.3 template<class Type, class Behavior = NullIsNull> mysqlpp::Null< Type, Behavior >::Null (const null_type &) `[inline]`

Construct a **Null** (p. 111) equal to SQL null.

This is typically used with the global `null` object. (Not to be confused with C's NULL type.) You can say something like...

```
Null<int> foo = null;
```

...to get a null `int`.

5.39.3 Member Function Documentation

5.39.3.1 template<class Type, class Behavior = NullIsNull> mysqlpp::Null< Type, Behavior >::operator Type & () `[inline]`

Converts this object to Type.

If is_null is set, returns whatever we consider that null "is", according to the Behavior parameter you used when instantiating this template. See **NullIsNull** (p. 118), **NullIsZero** (p. 119) and **NullIsBlank** (p. 117).

Otherwise, just returns the 'data' member.

References mysqlpp::Null< Type, Behavior >::data, and mysqlpp::Null< Type, Behavior >::is_null.

5.39.3.2 template<class Type, class Behavior = NullIsNull> Null& mysqlpp::Null< Type, Behavior >::operator= (const Type & *x*) `[inline]`

Assign a value to the object.

This marks the object as "not null" as a side effect.

References mysqlpp::Null< Type, Behavior >::data, and mysqlpp::Null< Type, Behavior >::is_null.

5.39.3.3 template<class Type, class Behavior = NullIsNull> Null& mysqlpp::Null< Type, Behavior >::operator= (const null_type & *n*) `[inline]`

Assign SQL null to this object.

This just sets the is_null flag; the data member is not affected until you call the Type() operator on it.

References mysqlpp::Null< Type, Behavior >::is_null.

5.39.3.4 template<class Type, class Behavior = NullIsNull> bool mysqlpp::Null< Type, Behavior >::operator== (const Null< Type > & *rhs*) const `[inline]`

Do equality comparison of two nullable values.

Two null objects are equal, and null is not equal to not-null. If neither is null, we delegate to operator == for the base data type.

References mysqlpp::Null< Type, Behavior >::data, and mysqlpp::Null< Type, Behavior >::is_null.

5.39.3.5 template<class Type, class Behavior = NullIsNull> bool mysqlpp::Null< Type, Behavior >::operator== (const null_type &) const `[inline]`

Do equality comparison against hard-coded SQL null.

This tells you the same thing as testing is_null member.

References mysqlpp::Null< Type, Behavior >::is_null.

5.39.3.6 template<class Type, class Behavior = NullIsNull> bool mysqlpp::Null< Type, Behavior >::operator< (const Null< Type > & *rhs*) const `[inline]`

Do less-than comparison of two nullable values.

Two null objects are equal to each other, and null is less than not-null. If neither is null, we delegate to operator < for the base data type.

References mysqlpp::Null< Type, Behavior >::data, and mysqlpp::Null< Type, Behavior >::is_null.

5.39.3.7 template<class Type, class Behavior = NullIsNull> bool mysqlpp::Null< Type, Behavior >::operator< (const null_type &) const `[inline]`

Do less-than comparison against hard-coded SQL null.

Always returns false because we can only be greater than or equal to a SQL null.

5.39.4 Member Data Documentation

5.39.4.1 template<class Type, class Behavior = NullIsNull> bool mysqlpp::Null< Type, Behavior >::is_null

If set, this object is considered equal to SQL null.

This flag affects how the Type() and << operators work.

Referenced by mysqlpp::Null< Type, Behavior >::operator Type &(), mysqlpp::Null< Type, Behavior >::operator<(), mysqlpp::Null< Type, Behavior >::operator=(), and mysqlpp::Null< Type, Behavior >::operator==().

The documentation for this class was generated from the following file:

- **null.h**

5.40 mysqlpp::null_type Class Reference

The type of the global mysqlpp::null object.

```
#include <null.h>
```

5.40.1 Detailed Description

The type of the global mysqlpp::null object.

User code shouldn't declare variables of this type. Use the **Null** (p. 111) template instead.

The documentation for this class was generated from the following file:

- **null.h**

5.41 mysqlpp::NullIsBlank Struct Reference

Class for objects that define SQL null as a blank C string.

```
#include <null.h>
```

5.41.1 Detailed Description

Class for objects that define SQL null as a blank C string.

Returns "" when you ask what null is, and is empty when you insert it into a C++ stream.

Used for the behavior parameter for template **Null** (p. 111)

The documentation for this struct was generated from the following file:

- **null.h**

5.42 mysqlpp::NullIsNull Struct Reference

Class for objects that define SQL null in terms of MySQL++'s **null_type** (p. 116).

```
#include <null.h>
```

5.42.1 Detailed Description

Class for objects that define SQL null in terms of MySQL++'s **null_type** (p. 116).

Returns a **null_type** (p. 116) instance when you ask what null is, and is "(NULL)" when you insert it into a C++ stream.

Used for the behavior parameter for template **Null** (p. 111)

The documentation for this struct was generated from the following file:

- **null.h**

5.43 mysqlpp::NullIsZero Struct Reference

Class for objects that define SQL null as 0.

`#include <null.h>`

5.43.1 Detailed Description

Class for objects that define SQL null as 0.

Returns 0 when you ask what null is, and is zero when you insert it into a C++ stream.

Used for the behavior parameter for template **Null** (p. 111)

The documentation for this struct was generated from the following file:

- **null.h**

5.44 mysqlpp::ObjectNotInitialized Class Reference

Exception (p. 84) thrown when you try to use an object that isn't completely initialized.

`#include <exceptions.h>`

Inheritance diagram for mysqlpp::ObjectNotInitialized:

Collaboration diagram for mysqlpp::ObjectNotInitialized:

Public Member Functions

- **ObjectNotInitialized** (const char ∗w="")

  Create exception object.

5.44.1 Detailed Description

Exception (p. 84) thrown when you try to use an object that isn't completely initialized.

The documentation for this class was generated from the following file:

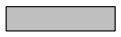
- **exceptions.h**

5.45 mysqlpp::Option Class Reference

Define abstract interface for all ∗Option subclasses.

```
#include <options.h>
```

Inheritance diagram for mysqlpp::Option:



Public Types

- enum **Error** { **err_NONE**, **err_api_limit**, **err_api_reject**, **err_connected** }

  Types of option setting errors we can diagnose.

Public Member Functions

- virtual ∼**Option** ()

Destroy object.

- virtual **Error** **set** (**DBDriver** ∗dbd)=0

  Apply option.

5.45.1 Detailed Description

Define abstract interface for all ∗Option subclasses.

This is the base class for the mid-level interface classes that take arguments, plus the direct base for options that take no arguments.

5.45.2 Member Enumeration Documentation

5.45.2.1 enum mysqlpp::Option::Error

Types of option setting errors we can diagnose.

Enumerator:

err_NONE	option was set successfully
err_api_limit	option not supported by underlying C API
err_api_reject	underlying C API returned error when setting option
err_connected	can't set the given option while connected

The documentation for this class was generated from the following file:

- **options.h**

5.46 mysqlpp::OptionalExceptions Class Reference

Interface allowing a class to have optional exceptions.

`#include <noexceptions.h>`

Inheritance diagram for mysqlpp::OptionalExceptions:

Public Member Functions

- **OptionalExceptions** (bool e=true)

  Default constructor.

- virtual ~**OptionalExceptions** ()

  Destroy object.

- void **enable_exceptions** () const

  Enable exceptions from the object.

- void **disable_exceptions** () const

  Disable exceptions from the object.

- bool **throw_exceptions** () const

  Returns true if exceptions are enabled.

Protected Member Functions

- void **set_exceptions** (bool e) const

  Sets the exception state to a particular value.

Friends

- class **NoExceptions**

  Declare ***NoExceptions*** (p. *108) to be our friend so it can access our protected functions.*

5.46.1 Detailed Description

Interface allowing a class to have optional exceptions.

A class derives from this one to acquire a standard interface for disabling exceptions, possibly only temporarily. By default, exceptions are enabled.

Note that all methods are const even though some of them change our internal flag indicating whether exceptions should be thrown. This is justifiable because this is just an interface class, and it changes the behavior of our subclass literally only in exceptional conditions. This Jesuitical interpretation of "const" is required because you may want to disable exceptions on const subclass instances.

If it makes you feel better about this, consider that the real change isn't within the const **OptionalExceptions** (p. 123) subclass instance. What changes is the code wrapping the method call on that instance that can optionally throw an exception. This outside code is in a better position to say what "const" means than the subclass instance.

5.46.2 Constructor & Destructor Documentation

5.46.2.1 mysqlpp::OptionalExceptions::OptionalExceptions (bool *e* = `true`) `[inline]`

Default constructor.

Parameters:

e if true, exceptions are enabled (this is the default)

5.46.3 Member Function Documentation

5.46.3.1 void mysqlpp::OptionalExceptions::set_exceptions (bool *e*) const `[inline, protected]`

Sets the exception state to a particular value.

This method is protected because it is only intended for use by subclasses' copy constructors and the like.

Referenced by mysqlpp::ResultBase::copy(), mysqlpp::Connection::copy(), and mysqlpp::Query::operator=().

The documentation for this class was generated from the following file:

- **noexceptions.h**

5.47 mysqlpp::Query Class Reference

A class for building and executing SQL queries.

`#include <query.h>`

Inheritance diagram for mysqlpp::Query:

Collaboration diagram for mysqlpp::Query:

Public Member Functions

- **Query** (**Connection** ∗c, bool te=true, const char ∗qstr=0)

  Create a new query object attached to a connection.

- **Query** (const **Query** &q)

  Create a new query object as a copy of another.

- ulonglong **affected_rows** ()

  Return the number of rows affected by the last query.

- size_t **escape_string** (std::string ∗ps, const char ∗original=0, size_t length=0) const

  Return a SQL-escaped version of a character buffer.

- size_t **escape_string** (char ∗escaped, const char ∗original, size_t length) const

    Return a SQL-escaped version of the given character buffer.

- int **errnum** () const

    Get the last error number that was set.

- const char ∗ **error** () const

    Get the last error message that was set.

- std::string **info** ()

    Returns information about the most recently executed query.

- ulonglong **insert_id** ()

    Get ID generated for an AUTO_INCREMENT column in the previous INSERT query.

- **Query** & **operator=** (const **Query** &rhs)

    Assign another query's state to this object.

- **operator void** ∗ () const

    Test whether the object has experienced an error condition.

- void **parse** ()

    Treat the contents of the query string as a template query.

- void **reset** ()

    Reset the query object so that it can be reused.

- std::string **str** ()

    Get built query as a C++ string.

- std::string **str** (const **SQLTypeAdapter** &arg0)

    Get built query as a C++ string with template query parameter substitution.

- std::string **str** (**SQLQueryParms** &p)

    Get built query as a null-terminated C++ string.

- bool **exec** ()

    Execute a built-up query.

- bool **exec** (const std::string &str)

    Execute a query.

- **SimpleResult execute** ()

  Execute built-up query.

- **SimpleResult execute** (**SQLQueryParms** &p)

  Execute template query using given parameters.

- **SimpleResult execute** (const **SQLTypeAdapter** &str)

  Execute a query that returns no rows.

- **SimpleResult execute** (const char ∗str, size_t len)

  Execute query in a known-length string of characters. This can include null characters.

- **UseQueryResult use** ()

  Execute a query that can return rows, with access to the rows in sequence.

- **UseQueryResult use** (**SQLQueryParms** &p)

  Execute a template query that can return rows, with access to the rows in sequence.

- **UseQueryResult use** (const **SQLTypeAdapter** &str)

  Execute a query that can return rows, with access to the rows in sequence.

- **UseQueryResult use** (const char ∗str, size_t len)

  Execute a query that can return rows, with access to the rows in sequence.

- **StoreQueryResult store** ()

  Execute a query that can return a result set.

- **StoreQueryResult store** (**SQLQueryParms** &p)

  Store results from a template query using given parameters.

- **StoreQueryResult store** (const **SQLTypeAdapter** &str)

  Execute a query that can return rows, returning all of the rows in a random-access container.

- **StoreQueryResult store** (const char ∗str, size_t len)

  Execute a query that can return rows, returning all of the rows in a random-access container.

- template<typename Function>
  Function **for_each** (const **SQLTypeAdapter** &query, Function fn)

  Execute a query, and call a functor for each returned row.

- template<typename Function>
  Function **for_each** (Function fn)

  Execute the query, and call a functor for each returned row.

- template<class SSQLS, typename Function>
  Function **for_each** (const SSQLS &ssqls, Function fn)

  Run a functor for every row in a table.

- template<class Sequence, typename Function>
  Function **store_if** (Sequence &con, const **SQLTypeAdapter** &query, Function fn)

  Execute a query, conditionally storing each row in a container.

- template<class Sequence, class SSQLS, typename Function>
  Function **store_if** (Sequence &con, const SSQLS &ssqls, Function fn)

  Pulls every row in a table, conditionally storing each one in a container.

- template<class Sequence, typename Function>
  Function **store_if** (Sequence &con, Function fn)

  Execute the query, conditionally storing each row in a container.

- **StoreQueryResult store_next** ()

  Return next result set, when processing a multi-query.

- bool **more_results** ()

  Return whether more results are waiting for a multi-query or stored procedure response.

- template<class Sequence>
  void **storein_sequence** (Sequence &con)

  Execute a query, storing the result set in an STL sequence container.

- template<class Sequence>
  void **storein_sequence** (Sequence &con, const **SQLTypeAdapter** &s)

  Executes a query, storing the result rows in an STL sequence container.

- template<class Seq>
  void **storein_sequence** (Seq &con, **SQLQueryParms** &p)

  Execute template query using given parameters, storing.

- template<class Set>
  void **storein_set** (**Set** &con)

Execute a query, storing the result set in an STL associative container.

- template<class Set>
  void **storein_set** (**Set** &con, const **SQLTypeAdapter** &s)

  Executes a query, storing the result rows in an STL set-associative container.

- template<class Set>
  void **storein_set** (**Set** &con, **SQLQueryParms** &p)

  Execute template query using given parameters, storing.

- template<class Container>
  void **storein** (Container &con)

  Execute a query, and store the entire result set in an STL container.

- template<class T>
  void **storein** (std::vector< T > &con, const **SQLTypeAdapter** &s)

  *Specialization of **storein_sequence()** (p. 146) for `std::vector`.*

- template<class T>
  void **storein** (std::deque< T > &con, const **SQLTypeAdapter** &s)

  *Specialization of **storein_sequence()** (p. 146) for `std::deque`.*

- template<class T>
  void **storein** (std::list< T > &con, const **SQLTypeAdapter** &s)

  *Specialization of **storein_sequence()** (p. 146) for `std::list`.*

- template<class T>
  void **storein** (std::set< T > &con, const **SQLTypeAdapter** &s)

  *Specialization of **storein_set()** (p. 147) for `std::set`.*

- template<class T>
  void **storein** (std::multiset< T > &con, const **SQLTypeAdapter** &s)

  *Specialization of **storein_set()** (p. 147) for `std::multiset`.*

- template<class T>
  Query & **update** (const T &o, const T &n)

  Replace an existing row's data with new data.

- template<class T>
  Query & **insert** (const T &v)

  Insert a new row.

- template<class Iter>
  Query & **insert** (Iter first, Iter last)

Insert multiple new rows.

- template<class T>
  Query & **replace** (const T &v)

  Insert new row unless there is an existing row that matches on a unique index, in which case we replace it.

Public Attributes

- **SQLQueryParms template_defaults**

  The default template parameters.

Friends

- class **SQLQueryParms**

5.47.1 Detailed Description

A class for building and executing SQL queries.

One does not generally create **Query** (p. 126) objects directly. Instead, call **mysqlpp::Connection::query()** (p. 40) to get one tied to that connection.

There are several ways to build and execute SQL queries with this class.

The way most like other database libraries is to pass a SQL statement in either the form of a C or C++ string to one of the **exec∗(),** (p. 137) **store∗(),** (p. 141) or **use()** (p. 139) methods. The query is executed immediately, and any results returned.

For more complicated queries, it's often more convenient to build up the query string over several C++ statements using Query's stream interface. It works like any other C++ stream (`std::cout`, `std::ostringstream`, etc.) in that you can just insert things into the stream, building the query up piece by piece. When the query string is complete, you call the overloaded version of **exec∗(),** (p. 137) **store∗(),** (p. 141) or **use()** (p. 139) takes no parameters, which executes the built query and returns any results.

If you are using the library's Specialized SQL Structures feature, **Query** (p. 126) has several special functions for generating common SQL queries from those structures. For instance, it offers the **insert()** (p. 149) method, which builds an INSERT query to add the contents of the SSQLS to the database. As with the stream interface, these methods only build the query string; call one of the parameterless methods mentioned previously to actually execute the query.

Finally, you can build "template queries". This is something like C's `printf()` function, in that you insert a specially-formatted query string into the object which contains placeholders for data. You call the **parse()** (p. 135) method to tell the **Query** (p. 126) object that the query string contains placeholders. Having done that, you call one of the the many **exec∗(),** (p. 138) **store∗(),** (p. 142) or **use()** (p. 140) overloads that take **SQLTypeAdapter** (p. 198) objects. There are 25 of each by default, differing only in the number of STA objects they take. (See `lib/querydef.pl` if you need to change the limit, or `examples/tquery2.cpp` for a way around it that doesn't require changing the library.) Only the version taking a single STA object is documented below, as to document all of them would just be repetitive. For each **Query** (p. 126) method that takes a single STA object, there's a good chance there's a set of undocumented overloads that take more of them for the purpose of filling out a template query.

See the user manual for more details about these options.

5.47.2 Constructor & Destructor Documentation

5.47.2.1 mysqlpp::Query::Query (Connection ∗ *c*, bool *te* = `true`, const char ∗ *qstr* = `0`)

Create a new query object attached to a connection.

This is the constructor used by **mysqlpp::Connection::query()** (p. 40).

Parameters:

c connection the finished query should be sent out on

te if true, throw exceptions on errors

qstr an optional initial query string

5.47.2.2 mysqlpp::Query::Query (const Query & *q*)

Create a new query object as a copy of another.

This is **not** a traditional copy ctor! Its only purpose is to make it possible to assign the return of **Connection::query()** (p. 40) to an empty **Query** (p. 126) object. In particular, the stream buffer and template query stuff will be empty in the copy, regardless of what values they have in the original.

5.47.3 Member Function Documentation

5.47.3.1 size_t mysqlpp::Query::escape_string (std::string ∗ *ps*, const char ∗ *original* = 0, size_t *length* = 0) const

Return a SQL-escaped version of a character buffer.

Parameters:

ps pointer to C++ string to hold escaped version; if original is 0, also holds the original data to be escaped

original if given, pointer to the character buffer to escape instead of contents of ∗ps

length if both this and original are given, number of characters to escape instead of ps->length()

Return values:

number of characters placed in ∗ps

This method has three basic operation modes:

- Pass just a pointer to a C++ string containing the original data to escape, plus act as receptacle for escaped version
- Pass a pointer to a C++ string to receive escaped string plus a pointer to a C string to be escaped
- Pass nonzero for all parameters, taking original to be a pointer to an array of char with given length; does not treat null characters as special

There's a degenerate fourth mode, where ps is zero: simply returns 0, because there is nowhere to store the result.

Note that if original is 0, we always ignore the length parameter even if it is nonzero. Length always comes from ps->length() in this case.

ps is a pointer because if it were a reference, the other overload would be impossible to call: the compiler would complain that the two overloads are ambiguous because std::string has a char∗ conversion ctor. A nice bonus is that pointer syntax makes it clearer that the first parameter is an "out" parameter.

See also:

comments for escape_string(char∗, const char∗, size_t) for further details.

Referenced by mysqlpp::SQLQueryParms::escape_string().

5.47.3.2 size_t mysqlpp::Query::escape_string (char ∗ *escaped*, const char ∗ *original*, size_t *length*) const

Return a SQL-escaped version of the given character buffer.

Parameters:

escaped character buffer to hold escaped version; must point to at least (length ∗ 2 + 1) bytes

original pointer to the character buffer to escape

length number of characters to escape

Return values:

number of characters placed in escaped

This is part of **Query** (p. 126) because proper SQL escaping takes the database's current character set into account, which requires access to the **Connection** (p. 32) object the query will go out on. Also, this function is very important to MySQL++'s **Query** (p. 126) stream manipulator mechanism, so it's more convenient for this method to live in **Query** (p. 126) rather than **Connection** (p. 32).

References mysqlpp::Connection::driver(), mysqlpp::DBDriver::escape_string(), and mysqlpp::DBDriver::escape_string_no_conn().

5.47.3.3 int mysqlpp::Query::errnum () const

Get the last error number that was set.

This just delegates to **Connection::errnum()** (p. 33). **Query** (p. 126) has nothing extra to say, so use either, as makes sense in your program.

References mysqlpp::Connection::errnum().

Referenced by exec(), execute(), store(), store_next(), and use().

5.47.3.4 const char ∗ mysqlpp::Query::error () const

Get the last error message that was set.

This just delegates to **Connection::error()** (p. 38). **Query** (p. 126) has nothing extra to say, so use either, as makes sense in your program.

References mysqlpp::Connection::error().

Referenced by exec(), execute(), store(), store_next(), and use().

5.47.3.5 ulonglong mysqlpp::Query::insert_id ()

Get ID generated for an AUTO_INCREMENT column in the previous INSERT query.

Return values:

0 if the previous query did not generate an ID. Use the SQL function LAST_-INSERT_ID() if you need the last ID generated by any query, not just the previous one.

References mysqlpp::Connection::driver(), and mysqlpp::DBDriver::insert_id().

Referenced by execute().

5.47.3.6 Query & mysqlpp::Query::operator= (const Query & *rhs*)

Assign another query's state to this object.

The same caveats apply to this operator as apply to the copy ctor.

References conn_, copacetic_, mysqlpp::OptionalExceptions::set_exceptions(), template_defaults, and mysqlpp::OptionalExceptions::throw_exceptions().

5.47.3.7 mysqlpp::Query::operator void ∗ () const

Test whether the object has experienced an error condition.

Allows for code constructs like this:

```
Query q = conn.query();
.... use query object
if (q) {
    ... no problems in using query object
}
else {
    ... an error has occurred
}
```

This method returns false if either the **Query** (p. 126) object or its associated **Connection** (p. 32) object has seen an error condition since the last operation.

5.47.3.8 void mysqlpp::Query::parse ()

Treat the contents of the query string as a template query.

This method sets up the internal structures used by all of the other members that accept template query parameters. See the "Template Queries" chapter in the user manual for more information.

References str().

5.47.3.9 void mysqlpp::Query::reset ()

Reset the query object so that it can be reused.

As of v3.0, **Query** (p. 126) objects auto-reset upon query execution unless you've set it up for making template queries. (It can't auto-reset in that situation, because it would forget the template info.) Therefore, the only time you must call this is if you have a **Query** (p. 126) object set up for making template queries, then want to build queries using one of the other methods. (Static strings, SSQLS, or the stream interface.)

References mysqlpp::SQLQueryParms::clear(), and template_defaults.

Referenced by exec(), execute(), store(), and use().

5.47.3.10 std::string mysqlpp::Query::str (const SQLTypeAdapter & *arg0*) `[inline]`

Get built query as a C++ string with template query parameter substitution.

Parameters:

arg0 the value to substitute for the first template query parameter; because **SQLTypeAdapter** (p. 198) implicitly converts from many different data types, this method is very flexible in what it accepts as a parameter. You shouldn't have to use the **SQLTypeAdapter** (p. 198) data type directly in your code.

There many more overloads of this type (25 total, by default; see `lib/querydef.pl`), each taking one more **SQLTypeAdapter** (p. 198) object than the previous one. See the template query overview above for more about this topic.

5.47.3.11 std::string mysqlpp::Query::str (SQLQueryParms & *p*)

Get built query as a null-terminated C++ string.

Parameters:

p template query parameters to use, overriding the ones this object holds, if any

5.47.3.12 bool mysqlpp::Query::exec () `[inline]`

Execute a built-up query.

Same as **exec()** (p. 136), except that it uses the query string built up within the query object already instead of accepting a query string from the caller.

Returns:

true if query was executed successfully

See also:

exec(const std::string& str) (p. 137), **execute()** (p. 137), **store()** (p. 141), **storein()** (p. 148), and **use()** (p. 139)

Referenced by mysqlpp::Connection::create_db(), and mysqlpp::Connection::drop_-db().

5.47.3.13 bool mysqlpp::Query::exec (const std::string & *str*)

Execute a query.

Same as **execute()** (p. 137), except that it only returns a flag indicating whether the query succeeded or not. It is basically a thin wrapper around the C API function `mysql_real_query()`.

Parameters:

str the query to execute

Returns:

true if query was executed successfully

See also:

execute() (p. 137), **store()** (p. 141), **storein()** (p. 148), and **use()** (p. 139)

References mysqlpp::Connection::driver(), errnum(), error(), mysqlpp::DBDriver::execute(), reset(), and mysqlpp::OptionalExceptions::throw_-exceptions().

5.47.3.14 SimpleResult mysqlpp::Query::execute () `[inline]`

Execute built-up query.

Use one of the **execute()** (p. 137) overloads if you don't expect the server to return a result set. For instance, a DELETE query. The returned **SimpleResult** (p. 187) object contains status information from the server, such as whether the query succeeded, and if so how many rows were affected.

This overloaded version of **execute()** (p. 137) simply executes the query that you have built up in the object in some way. (For instance, via the **insert()** (p. 149) method, or by using the object's stream interface.)

Returns:

SimpleResult (p. 187) status information about the query

See also:

exec() (p. 136), **store()** (p. 141), **storein()** (p. 148), and **use()** (p. 139)

Referenced by mysqlpp::Transaction::commit(), execute(), and mysqlpp::Transaction::rollback().

5.47.3.15 SimpleResult mysqlpp::Query::execute (SQLQueryParms & *p*)

Execute template query using given parameters.

This method should only be used by code that doesn't know, at compile time, how many parameters it will have. This is useful within the library, and also for code that builds template queries dynamically, at run time.

Parameters:

p parameters to use in the template query.

References execute(), mysqlpp::SQLQueryParms::processing_, str(), and template_-defaults.

5.47.3.16 SimpleResult mysqlpp::Query::execute (const SQLTypeAdapter & *str*)

Execute a query that returns no rows.

Parameters:

str if this object is set up as a template query, this is the value to substitute for the first template query parameter; else, it is the SQL query string to execute

Because **SQLTypeAdapter** (p. 198) can be initialized from either a C string or a C++ string, this overload accepts query strings in either form. Beware, **SQLTypeAdapter** (p. 198) also accepts many other data types (this is its *raison d'etre*), so it will let you write code that compiles but results in bogus SQL queries.

To support template queries, there many more overloads of this type (25 total, by default; see `lib/querydef.pl`), each taking one more **SQLTypeAdapter** (p. 198) object than the previous one. See the template query overview above for more about this topic.

References mysqlpp::SQLTypeAdapter::data(), execute(), mysqlpp::SQLTypeAdapter::length(), mysqlpp::SQLQueryParms::processing_, and template_defaults.

5.47.3.17 SimpleResult mysqlpp::Query::execute (const char ∗ *str*, size_t *len*)

Execute query in a known-length string of characters. This can include null characters.

Executes the query immediately, and returns the results.

References affected_rows(), mysqlpp::Connection::driver(), errnum(), error(), mysqlpp::DBDriver::execute(), info(), insert_id(), reset(), and mysqlpp::OptionalExceptions::throw_exceptions().

5.47.3.18 UseQueryResult mysqlpp::Query::use () `[inline]`

Execute a query that can return rows, with access to the rows in sequence.

Use one of the **use()** (p. 139) overloads if memory efficiency is important. They return an object that can walk through the result records one by one, without fetching the entire result set from the server. This is superior to **store()** (p. 141) when there are a large number of results; **store()** (p. 141) would have to allocate a large block of memory to hold all those records, which could cause problems.

A potential downside of this method is that MySQL database resources are tied up until the result set is completely consumed. Do your best to walk through the result set as expeditiously as possible.

The name of this method comes from the MySQL C API function that initiates the retrieval process, `mysql_use_result()`. This method is implemented in terms of that function.

This function has the same set of overloads as **execute()** (p. 137).

Returns:

UseQueryResult (p. 247) object that can walk through result set serially

See also:

exec() (p. 136), **execute()** (p. 137), **store()** (p. 141) and **storein()** (p. 148)

Referenced by use().

5.47.3.19 UseQueryResult mysqlpp::Query::use (SQLQueryParms & *p*)

Execute a template query that can return rows, with access to the rows in sequence.

This method should only be used by code that doesn't know, at compile time, how many parameters it will have. This is useful within the library, and also for code that builds template queries dynamically, at run time.

Parameters:

p parameters to use in the template query.

References mysqlpp::SQLQueryParms::processing_, str(), template_defaults, and use().

5.47.3.20 UseQueryResult mysqlpp::Query::use (const SQLTypeAdapter & *str*)

Execute a query that can return rows, with access to the rows in sequence.

Parameters:

str if this object is set up as a template query, this is the value to substitute for the first template query parameter; else, it is the SQL query string to execute

Because **SQLTypeAdapter** (p. 198) can be initialized from either a C string or a C++ string, this overload accepts query strings in either form. Beware, **SQLTypeAdapter** (p. 198) also accepts many other data types (this is its *raison d'etre*), so it will let you write code that compiles but results in bogus SQL queries.

To support template queries, there many more overloads of this type (25 total, by default; see `lib/querydef.pl`), each taking one more **SQLTypeAdapter** (p. 198) object than the previous one. See the template query overview above for more about this topic.

References mysqlpp::SQLTypeAdapter::data(), mysqlpp::SQLTypeAdapter::length(), mysqlpp::SQLQueryParms::processing_, template_defaults, and use().

5.47.3.21 UseQueryResult mysqlpp::Query::use (const char ∗ *str*, size_t *len*)

Execute a query that can return rows, with access to the rows in sequence.

This overload is for situations where you have the query in a C string and have its length already. If you want to execute a query in a null-terminated C string or have the query string in some other form, you probably want to call **use(const SQLTypeAdapter&)** (p. 140) instead. **SQLTypeAdapter** (p. 198) converts from plain C strings and other useful data types implicitly.

References mysqlpp::Connection::driver(), errnum(), mysqlpp::Connection::errnum(), error(), mysqlpp::DBDriver::execute(), reset(), mysqlpp::OptionalExceptions::throw_-exceptions(), and mysqlpp::DBDriver::use_result().

5.47.3.22 StoreQueryResult mysqlpp::Query::store () `[inline]`

Execute a query that can return a result set.

Use one of the **store()** (p. 141) overloads to execute a query and retrieve the entire result set into memory. This is useful if you actually need all of the records at once, but if not, consider using one of the **use()** (p. 139) methods instead, which returns the results one at a time, so they don't allocate as much memory as **store()** (p. 141).

You must use **store()** (p. 141), **storein()** (p. 148) or **use()** (p. 139) for `SELECT`, `SHOW`, `DESCRIBE` and `EXPLAIN` queries. You can use these functions with other query types, but since they don't return a result set, **exec()** (p. 136) and **execute()** (p. 137) are more efficient.

The name of this method comes from the MySQL C API function it is implemented in terms of, `mysql_store_result()`.

This function has the same set of overloads as **execute()** (p. 137).

Returns:

StoreQueryResult (p. 209) object containing entire result set

See also:

exec() (p. 136), **execute()** (p. 137), **storein()** (p. 148), and **use()** (p. 139)

Referenced by mysqlpp::Connection::count_rows(), store(), and store_next().

5.47.3.23 StoreQueryResult mysqlpp::Query::store (SQLQueryParms & *p*)

Store results from a template query using given parameters.

This method should only be used by code that doesn't know, at compile time, how many parameters it will have. This is useful within the library, and also for code that builds template queries dynamically, at run time.

Parameters:

p parameters to use in the template query.

References mysqlpp::SQLQueryParms::processing_, store(), str(), and template_-defaults.

5.47.3.24 StoreQueryResult mysqlpp::Query::store (const SQLTypeAdapter & *str*)

Execute a query that can return rows, returning all of the rows in a random-access container.

Parameters:

str if this object is set up as a template query, this is the value to substitute for the first template query parameter; else, it is the SQL query string to execute

Because **SQLTypeAdapter** (p. 198) can be initialized from either a C string or a C++ string, this overload accepts query strings in either form. Beware, **SQLTypeAdapter** (p. 198) also accepts many other data types (this is its *raison d'etre*), so it will let you write code that compiles but results in bogus SQL queries.

To support template queries, there many more overloads of this type (25 total, by default; see `lib/querydef.pl`), each taking one more **SQLTypeAdapter** (p. 198) object than the previous one. See the template query overview above for more about this topic.

References mysqlpp::SQLTypeAdapter::data(), mysqlpp::SQLTypeAdapter::length(), mysqlpp::SQLQueryParms::processing_, store(), and template_defaults.

5.47.3.25 StoreQueryResult mysqlpp::Query::store (const char ∗ *str*, size_t *len*)

Execute a query that can return rows, returning all of the rows in a random-access container.

This overload is for situations where you have the query in a C string and have its length already. If you want to execute a query in a null-terminated C string or have the query string in some other form, you probably want to call **store(const SQLTypeAdapter&)** (p. 142) instead. **SQLTypeAdapter** (p. 198) converts from plain C strings and other useful data types implicitly.

References mysqlpp::Connection::driver(), errnum(), mysqlpp::Connection::errnum(), error(), mysqlpp::DBDriver::execute(), reset(), mysqlpp::DBDriver::store_result(), and mysqlpp::OptionalExceptions::throw_exceptions().

5.47.3.26 template<typename Function> Function mysqlpp::Query::for_each (const SQLTypeAdapter & *query*, Function *fn*) `[inline]`

Execute a query, and call a functor for each returned row.

This method wraps a **use()** (p. 139) query, calling the given functor for every returned row. It is analogous to STL's **for_each()** (p. 142) algorithm, but instead of iterating over some range within a container, it iterates over a result set produced by a query.

Parameters:

query the query string

fn the functor called for each row

Returns:

a copy of the passed functor

References mysqlpp::UseQueryResult::fetch_row().

5.47.3.27 template<typename Function> Function mysqlpp::Query::for_each (Function *fn*) `[inline]`

Execute the query, and call a functor for each returned row.

Just like **for_each(const SQLTypeAdapter&, Function)** (p. 142), but it uses the query string held by the **Query** (p. 126) object already

Parameters:

fn the functor called for each row

Returns:

a copy of the passed functor

References mysqlpp::UseQueryResult::fetch_row().

5.47.3.28 template<class SSQLS, typename Function> Function mysqlpp::Query::for_each (const SSQLS & *ssqls*, Function *fn*) `[inline]`

Run a functor for every row in a table.

Just like **for_each(Function)** (p. 143), except that it builds a "select $*$ from TABLE" query using the SQL table name from the SSQLS instance you pass.

Parameters:

ssqls the SSQLS instance to get a table name from

fn the functor called for each row

Returns:

a copy of the passed functor

References mysqlpp::UseQueryResult::fetch_row().

5.47.3.29 template<class Sequence, typename Function> Function mysqlpp::Query::store_if (Sequence & *con*, const SQLTypeAdapter & *query*, Function *fn*) [inline]

Execute a query, conditionally storing each row in a container.

This method wraps a **use()** (p. 139) query, calling the given functor for every returned row, and storing the results in the given sequence container if the functor returns true.

This is analogous to the STL copy_if() algorithm, except that the source rows come from a database query instead of another container. (copy_if() isn't a standard STL algorithm, but only due to an oversight by the standardization committee.) This fact may help you to remember the order of the parameters: the container is the destination, the query is the source, and the functor is the predicate; it's just like an STL algorithm.

Parameters:

con the destination container; needs a push_back() method

query the query string

fn the functor called for each row

Returns:

a copy of the passed functor

References mysqlpp::UseQueryResult::fetch_row().

5.47.3.30 template<class Sequence, class SSQLS, typename Function> Function mysqlpp::Query::store_if (Sequence & *con*, const SSQLS & *ssqls*, Function *fn*) [inline]

Pulls every row in a table, conditionally storing each one in a container.

Just like **store_if(Sequence&, const SQLTypeAdapter&, Function)** (p. 144), but it uses the SSQLS instance to construct a "select $*$ from TABLE" query, using the table name field in the SSQLS.

Parameters:

con the destination container; needs a push_back() method

ssqls the SSQLS instance to get a table name from

fn the functor called for each row

Returns:

a copy of the passed functor

References mysqlpp::UseQueryResult::fetch_row().

5.47.3.31 template<class Sequence, typename Function> Function mysqlpp::Query::store_if (Sequence & *con*, Function *fn*) `[inline]`

Execute the query, conditionally storing each row in a container.

Just like **store_if(Sequence&, const SQLTypeAdapter&, Function)** (p. 144), but it uses the query string held by the **Query** (p. 126) object already

Parameters:

con the destination container; needs a push_back() method

fn the functor called for each row

Returns:

a copy of the passed functor

References mysqlpp::UseQueryResult::fetch_row().

5.47.3.32 StoreQueryResult mysqlpp::Query::store_next ()

Return next result set, when processing a multi-query.

There are two cases where you'd use this function instead of the regular **store()** (p. 141) functions.

First, when handling the result of executing multiple queries at once. (See `this page` in the MySQL documentation for details.)

Second, when calling a stored procedure, MySQL can return the result as a set of results.

In either case, you must consume all results before making another MySQL query, even if you don't care about the remaining results or result sets.

As the MySQL documentation points out, you must set the MYSQL_OPTION_-MULTI_STATEMENTS_ON flag on the connection in order to use this feature. See **Connection::set_option()** (p. 40).

Multi-queries only exist in MySQL v4.1 and higher. Therefore, this function just wraps **store()** (p. 141) when built against older API libraries.

Returns:

StoreQueryResult (p. 209) object containing the next result set.

References mysqlpp::Connection::driver(), errnum(), mysqlpp::Connection::errnum(), error(), mysqlpp::DBDriver::next_result(), mysqlpp::DBDriver::nr_error, mysqlpp::DBDriver::nr_more_results, store(), mysqlpp::DBDriver::store_result(), and mysqlpp::OptionalExceptions::throw_exceptions().

5.47.3.33 bool mysqlpp::Query::more_results ()

Return whether more results are waiting for a multi-query or stored procedure response.

If this function returns true, you must call **store_next()** (p. 145) to fetch the next result set before you can execute more queries.

Wraps mysql_more_results() in the MySQL C API. That function only exists in MySQL v4.1 and higher. Therefore, this function always returns false when built against older API libraries.

Returns:

true if another result set exists

References mysqlpp::Connection::driver(), and mysqlpp::DBDriver::more_results().

5.47.3.34 template<class Sequence> void mysqlpp::Query::storein_sequence (Sequence & *con*) `[inline]`

Execute a query, storing the result set in an STL sequence container.

This function works much like **store()** (p. 141) from the caller's perspective, because it returns the entire result set at once. It's actually implemented in terms of **use()** (p. 139), however, so that memory for the result set doesn't need to be allocated twice.

There are many overloads for this function, pretty much the same as for **execute()** (p. 137), except that there is a Container parameter at the front of the list. So, you can pass a container and a query string, or a container and template query parameters.

Parameters:

con any STL sequence container, such as `std::vector`

See also:

exec() (p. 136), **execute()** (p. 137), **store()** (p. 141), and **use()** (p. 139)

5.47.3.35 template<class Sequence> void mysqlpp::Query::storein_sequence (Sequence & *con*, const SQLTypeAdapter & *s*) `[inline]`

Executes a query, storing the result rows in an STL sequence container.

Parameters:

con the container to store the results in

s if **Query** (p. 126) is set up as a template query, this is the value to substitute for the first template query parameter; else, the SQL query string

There many more overloads of this type (25 total, by default; see `lib/querydef.pl`), each taking one more **SQLTypeAdapter** (p. 198) object than the previous one. See the template query overview above for more about this topic.

References mysqlpp::UseQueryResult::fetch_lengths(), and mysqlpp::UseQueryResult::fetch_raw_row().

5.47.3.36 template<class Seq> void mysqlpp::Query::storein_sequence (Seq & *con*, SQLQueryParms & *p*) `[inline]`

Execute template query using given parameters, storing.

the results in a sequence type container.

This method should only be used by code that doesn't know, at compile time, how many parameters it will have. This is useful within the library, and also for code that builds template queries dynamically, at run time.

Parameters:

con container that will receive the results

p parameters to use in the template query.

5.47.3.37 template<class Set> void mysqlpp::Query::storein_set (Set & *con*) `[inline]`

Execute a query, storing the result set in an STL associative container.

The same thing as **storein_sequence()** (p. 146), except that it's used with associative STL containers, such as `std::set`. Other than that detail, that method's comments apply equally well to this one.

5.47.3.38 template<class Set> void mysqlpp::Query::storein_set (Set & *con*, const SQLTypeAdapter & *s*) `[inline]`

Executes a query, storing the result rows in an STL set-associative container.

Parameters:

con the container to store the results in

s if **Query** (p. 126) is set up as a template query, this is the value to substitute for the first template query parameter; else, the SQL query string

There many more overloads of this type (25 total, by default; see `lib/querydef.pl`), each taking one more **SQLTypeAdapter** (p. 198) object than the previous one. See the template query overview above for more about this topic.

References mysqlpp::UseQueryResult::fetch_lengths(), and mysqlpp::UseQueryResult::fetch_raw_row().

5.47.3.39 template<class Set> void mysqlpp::Query::storein_set (Set & *con*, SQLQueryParms & *p*) `[inline]`

Execute template query using given parameters, storing.

the results in a set type container.

This method should only be used by code that doesn't know, at compile time, how many parameters it will have. This is useful within the library, and also for code that builds template queries dynamically, at run time.

Parameters:

- ***con*** container that will receive the results
- ***p*** parameters to use in the template query.

5.47.3.40 template<class Container> void mysqlpp::Query::storein (Container & *con*) `[inline]`

Execute a query, and store the entire result set in an STL container.

This is a set of specialized template functions that call either **storein_sequence()** (p. 146) or **storein_set()** (p. 147), depending on the type of container you pass it. It understands `std::vector`, `deque`, `list`, `slist` (a common C++ library extension), `set`, and `multiset`.

Like the functions it wraps, this is actually an overloaded set of functions. See the other functions' documentation for details.

Use this function if you think you might someday switch your program from using a set-associative container to a sequence container for storing result sets, or vice versa.

See **exec()** (p. 136), **execute()** (p. 137), **store()** (p. 141), and **use()** (p. 139) for alternative query execution mechanisms.

5.47.3.41 template<class T> Query& mysqlpp::Query::update (const T & *o*, const T & *n*) `[inline]`

Replace an existing row's data with new data.

This function builds an UPDATE SQL query using the new row data for the SET clause, and the old row data for the WHERE clause. One uses it with MySQL++'s Specialized SQL Structures mechanism.

Parameters:

o old row

n new row

See also:

insert() (p. 149), **replace()** (p. 150)

5.47.3.42 template<class T> Query& mysqlpp::Query::insert (const T & *v*) `[inline]`

Insert a new row.

This function builds an INSERT SQL query. One uses it with MySQL++'s Specialized SQL Structures mechanism.

Parameters:

v new row

See also:

replace() (p. 150), **update()** (p. 148)

5.47.3.43 template<class Iter> Query& mysqlpp::Query::insert (Iter *first*, Iter *last*) `[inline]`

Insert multiple new rows.

Builds an INSERT SQL query using items from a range within an STL container. Insert the entire contents of the container by using the begin() and end() iterators of the container as parameters to this function.

Parameters:

first iterator pointing to first element in range to insert

last iterator pointing to one past the last element to insert

See also:

replace() (p. 150), **update()** (p. 148)

5.47.3.44 template<class T> Query& mysqlpp::Query::replace (const T & *v*) `[inline]`

Insert new row unless there is an existing row that matches on a unique index, in which case we replace it.

This function builds a REPLACE SQL query. One uses it with MySQL++'s Specialized SQL Structures mechanism.

Parameters:

v new row

See also:

insert() (p. 149), **update()** (p. 148)

5.47.4 Member Data Documentation

5.47.4.1 SQLQueryParms mysqlpp::Query::template_defaults

The default template parameters.

Used for filling in parameterized queries.

Referenced by execute(), operator=(), reset(), store(), and use().

The documentation for this class was generated from the following files:

- **query.h**
- query.cpp

5.48 mysqlpp::ReadDefaultFileOption Class Reference

Override use of my.cnf.

`#include <options.h>`

Inheritance diagram for mysqlpp::ReadDefaultFileOption:

Collaboration diagram for mysqlpp::ReadDefaultFileOption:

5.48.1 Detailed Description

Override use of my.cnf.

The documentation for this class was generated from the following file:

- **options.h**

5.49 mysqlpp::ReadDefaultGroupOption Class Reference

Override use of my.cnf.

`#include <options.h>`

Inheritance diagram for mysqlpp::ReadDefaultGroupOption:

Collaboration diagram for mysqlpp::ReadDefaultGroupOption:

5.49.1 Detailed Description

Override use of my.cnf.

The documentation for this class was generated from the following file:

- **options.h**

5.50 mysqlpp::ReadTimeoutOption Class Reference

Set (p. 182) timeout for IPC data reads.

```
#include <options.h>
```

Inheritance diagram for mysqlpp::ReadTimeoutOption:

Collaboration diagram for mysqlpp::ReadTimeoutOption:

5.50.1 Detailed Description

Set (p. 182) timeout for IPC data reads.

The documentation for this class was generated from the following file:

- **options.h**

5.51 mysqlpp::ReconnectOption Class Reference

Enable automatic reconnection to server.

`#include <options.h>`

Inheritance diagram for mysqlpp::ReconnectOption:

Collaboration diagram for mysqlpp::ReconnectOption:

5.51.1 Detailed Description

Enable automatic reconnection to server.

The documentation for this class was generated from the following file:

- **options.h**

5.52 mysqlpp::RefCountedPointer< T, Destroyer > Class Template Reference

Creates an object that acts as a reference-counted pointer to another object.

`#include <refcounted.h>`

Inheritance diagram for mysqlpp::RefCountedPointer< T, Destroyer >:

Collaboration diagram for mysqlpp::RefCountedPointer< T, Destroyer >:

Public Types

- typedef **RefCountedPointer**< T > **ThisType**

  alias for this object's type

Public Member Functions

- **RefCountedPointer** ()

  Default constructor.

- **RefCountedPointer** (T $*$c)

  Standard constructor.

- **RefCountedPointer** (const **ThisType** &other)

  Copy constructor.

- ∼**RefCountedPointer** ()

  Destructor.

- **ThisType** & **assign** (T ∗c)

  Sets (or resets) the pointer to the counted object.

- **ThisType** & **assign** (const **ThisType** &other)

  Copy an existing refcounted pointer.

- **ThisType** & **operator=** (T ∗c)

  ***Set** (p. 182) (or reset) the pointer to the counted object.*

- **ThisType** & **operator=** (const **ThisType** &rhs)

  Copy an existing refcounted pointer.

- T ∗ **operator** → () const

  Access the object through the smart pointer.

- T & **operator**∗ () const

  Dereference the smart pointer.

- **operator void** ∗ ()

  Returns the internal raw pointer converted to void∗.

- **operator const void** ∗ () const

  Returns the internal raw pointer converted to const void∗.

- T ∗ **raw** ()

  Return the raw pointer in T∗ context.

- const T ∗ **raw** () const

  Return the raw pointer when used in const T∗ context.

- void **swap** (**ThisType** &other)

  Exchange our managed memory with another pointer.

5.52.1 Detailed Description

template<class T, class Destroyer = RefCountedPointerDestroyer<T>> class mysqlpp::RefCountedPointer< T, Destroyer >

Creates an object that acts as a reference-counted pointer to another object.

Resulting type acts like a pointer in all respects, except that it manages the memory it points to by observing how many users there are for the object.

This attempts to be as automatic as reference counting in a programming language with memory management. Like all automatic memory management schemes, it has penalties: it turns the single indirection of an unmanaged pointer into a double indirection, and has additional management overhead in the assignment operators due to the reference counter. This is an acceptable tradeoff when wrapping objects that are expensive to copy, and which need to be "owned" by disparate parties: you can allocate the object just once, then pass around the reference counted pointer, knowing that the last user will "turn out the lights".

Implementation detail: You may notice that this class manages two pointers, one to the data we're managing, and one to the reference count. You might wonder why we don't wrap these up into a structure and keep just a pointer to an instance of it to simplify the memory management. It would indeed do that, but then every access to the data we manage would be a triple indirection instead of just double. It's a tradeoff, and we've chosen to take a minor complexity hit to avoid the performance hit.

5.52.2 Constructor & Destructor Documentation

5.52.2.1 template<class T, class Destroyer = RefCountedPointerDestroyer<T>> mysqlpp::RefCountedPointer< T, Destroyer >::RefCountedPointer () `[inline]`

Default constructor.

An object constructed this way is useless until you vivify it with **operator =()** (p. 158) or **assign()** (p. 158).

5.52.2.2 template<class T, class Destroyer = RefCountedPointerDestroyer<T>> mysqlpp::RefCountedPointer< T, Destroyer >::RefCountedPointer (T ∗ *c*) `[inline, explicit]`

Standard constructor.

Parameters:

c A pointer to the object to be managed. If you pass 0, it's like calling the default ctor instead, only more work: the object's useless until you vivify it with

operator =() (p. 158) or **assign()** (p. 158).

5.52.2.3 template<class T, class Destroyer = RefCountedPointerDestroyer<T>> mysqlpp::RefCountedPointer< T, Destroyer >::~RefCountedPointer () `[inline]`

Destructor.

This only destroys the managed memory if the reference count drops to 0.

5.52.3 Member Function Documentation

5.52.3.1 template<class T, class Destroyer = RefCountedPointerDestroyer<T>> ThisType& mysqlpp::RefCountedPointer< T, Destroyer >::assign (T * *c*) `[inline]`

Sets (or resets) the pointer to the counted object.

If we are managing a pointer, this decrements the refcount for it and destroys the managed object if the refcount falls to 0.

This is a no-op if you pass the same pointer we're already managing.

Referenced by mysqlpp::RefCountedPointer< mysqlpp::SQLBuffer >::operator=().

5.52.3.2 template<class T, class Destroyer = RefCountedPointerDestroyer<T>> ThisType& mysqlpp::RefCountedPointer< T, Destroyer >::assign (const ThisType & *other*) `[inline]`

Copy an existing refcounted pointer.

If we are managing a pointer, this decrements the refcount for it and destroys the managed object if the refcount falls to 0. Then we increment the other object's reference count and copy that refcount and the managed pointer into this object.

This is a no-op if you pass a reference to this same object.

5.52.3.3 template<class T, class Destroyer = RefCountedPointerDestroyer<T>> ThisType& mysqlpp::RefCountedPointer< T, Destroyer >::operator= (T * *c*) `[inline]`

Set (p. 182) (or reset) the pointer to the counted object.

This is essentially the same thing as **assign(T∗)** (p. 158). The choice between the two is just a matter of syntactic preference.

5.52.3.4 template<class T, class Destroyer = RefCountedPointerDestroyer<T>> ThisType& mysqlpp::RefCountedPointer< T, Destroyer >::operator= (const ThisType & *rhs*) `[inline]`

Copy an existing refcounted pointer.

This is essentially the same thing as **assign(const ThisType&)** (p. 158). The choice between the two is just a matter of syntactic preference.

5.52.3.5 template<class T, class Destroyer = RefCountedPointerDestroyer<T>> mysqlpp::RefCountedPointer< T, Destroyer >::operator void ∗ () `[inline]`

Returns the internal raw pointer converted to void∗.

This isn't intended to be used directly; if you need the pointer, call **raw()** (p. 156) instead. It's used internally by the compiler to implement operators bool, ==, and !=

WARNING: This makes it possible to say

```
RefCountedPointer<Foo> bar(new Foo);
delete bar;
```

This will almost kinda sorta do the right thing: the Foo object held by the refcounted pointer will be destroyed as you wanted, but then when the refcounted pointer goes out of scope, the memory is deleted a second time, which will probably crash your program. This is easy to accidentally do when converting a good ol' unmanaged pointer to a refcounted pointer and forgetting to remove the delete calls needed previously.

5.52.3.6 template<class T, class Destroyer = RefCountedPointerDestroyer<T>> mysqlpp::RefCountedPointer< T, Destroyer >::operator const void ∗ () const `[inline]`

Returns the internal raw pointer converted to const void∗.

See also:

comments for operator void∗()

The documentation for this class was generated from the following file:

- **refcounted.h**

5.53 mysqlpp::RefCountedPointerDestroyer< T > Struct Template Reference

Functor to call delete on the pointer you pass to it.

```
#include <refcounted.h>
```

Public Member Functions

- void **operator()** (T ∗doomed) const

  Functor implementation.

5.53.1 Detailed Description

template<class T> struct mysqlpp::RefCountedPointerDestroyer< T >

Functor to call delete on the pointer you pass to it.

The default "destroyer" for **RefCountedPointer** (p. 155). You won't use this directly, you'll pass a functor of your own devising for the second parameter to the **RefCountedPointer** (p. 155) template to override this. Or simpler, just specialize this template for your type if possible: see ResUse::result_.

The documentation for this struct was generated from the following file:

- **refcounted.h**

5.54 mysqlpp::RefCountedPointerDestroyer< MYSQL_RES > Struct Template Reference

Functor to call mysql_free_result() on the pointer you pass to it.

`#include <result.h>`

Public Member Functions

- void **operator()** (MYSQL_RES ∗doomed) const

  Functor implementation.

5.54.1 Detailed Description

template<> struct mysqlpp::RefCountedPointerDestroyer< MYSQL_RES >

Functor to call mysql_free_result() on the pointer you pass to it.

This overrides RefCountedPointer's default destroyer, which uses operator delete; it annoys the C API when you nuke its data structures this way. :)

The documentation for this struct was generated from the following file:

- **result.h**

5.55 mysqlpp::ReportDataTruncationOption Class Reference

Set (p. 182) reporting of data truncation errors.

`#include <options.h>`

Inheritance diagram for mysqlpp::ReportDataTruncationOption:

Collaboration diagram for mysqlpp::ReportDataTruncationOption:

5.55.1 Detailed Description

Set (p. 182) reporting of data truncation errors.

The documentation for this class was generated from the following file:

- **options.h**

5.56 mysqlpp::ResultBase Class Reference

Base class for **StoreQueryResult** (p. 209) and **UseQueryResult** (p. 247).

`#include <result.h>`

Inheritance diagram for mysqlpp::ResultBase:

Collaboration diagram for mysqlpp::ResultBase:

Public Member Functions

- virtual ~**ResultBase** ()

    Destroy object.

- const **Field** & **fetch_field** () const

    Returns the next field in this result set.

- const **Field** & **fetch_field** (Fields::size_type i) const

    Returns the given field in this result set.

- const **Field** & **field** (unsigned int i) const

    Get the underlying ***Field*** *(p. 87) structure given its index.*

- const Fields & **fields** () const

    Get the underlying Fields structure.

- const std::string & **field_name** (int i) const

Get the name of the field at the given index.

- const **RefCountedPointer**< **FieldNames** > & **field_names** () const

  Get the names of the fields within this result set.

- int **field_num** (const std::string &) const

  Get the index of the named field.

- const FieldTypes::value_type & **field_type** (int i) const

  Get the type of a particular field within this result set.

- const **RefCountedPointer**< **FieldTypes** > & **field_types** () const

  Get a list of the types of the fields within this result set.

- size_t **num_fields** () const

  Returns the number of fields in this result set.

- const char ∗ **table** () const

  Return the name of the table the result set comes from.

Protected Member Functions

- **ResultBase** ()

  Create empty object.

- **ResultBase** (MYSQL_RES ∗result, **DBDriver** ∗dbd, bool te=true)

  Create the object, fully initialized.

- **ResultBase** (const **ResultBase** &other)

  Create object as a copy of another ***ResultBase*** (p. *163).*

- **ResultBase** & **copy** (const **ResultBase** &other)

  Copy another ***ResultBase*** (p. *163) object's contents into this one.*

Protected Attributes

- **DBDriver** ∗ **driver_**

  Access to DB driver; fully initted if nonzero.

- Fields **fields_**

list of fields in result

- **RefCountedPointer< FieldNames > names_**
  list of field names in result
- **RefCountedPointer< FieldTypes > types_**
  list of field types in result
- Fields::size_type **current_field_**
  Default field index used by ***fetch_field()*** *(*p. *163).*

5.56.1 Detailed Description

Base class for **StoreQueryResult** (p. 209) and **UseQueryResult** (p. 247).

Not useful directly. Just contains common functionality for its subclasses.

5.56.2 Member Function Documentation

5.56.2.1 int mysqlpp::ResultBase::field_num (const std::string & *i*) const

Get the index of the named field.

This is the inverse of **field_name()** (p. 163).

References names_, and mysqlpp::OptionalExceptions::throw_exceptions().

5.56.3 Member Data Documentation

5.56.3.1 Fields::size_type mysqlpp::ResultBase::current_field_ `[mutable, protected]`

Default field index used by **fetch_field()** (p. 163).

It's mutable because it's just internal housekeeping: it's changed by **fetch_field(void)** (p. 163), but it doesn't change the "value" of the result. See mutability justification for UseQueryResult::result_: this field provides functionality we used to get through result_, so it's relevant here, too.

Referenced by copy().

The documentation for this class was generated from the following files:

- **result.h**
- result.cpp

5.57 mysqlpp::Row Class Reference

Manages rows from a result set.

`#include <row.h>`

Inheritance diagram for mysqlpp::Row:

Collaboration diagram for mysqlpp::Row:

Public Types

- typedef std::vector< **String** > **list_type**

  type of our internal data list

- typedef list_type::const_iterator **const_iterator**

  constant iterator type

- typedef list_type::const_reference **const_reference**

  constant reference type

- typedef list_type::const_reverse_iterator **const_reverse_iterator**

  const reverse iterator type

- typedef list_type::difference_type **difference_type**

  type for index differences

- typedef **const_iterator iterator**

  iterator type

- typedef **const_reference reference**

reference type

- typedef **const_reverse_iterator reverse_iterator**

  mutable reverse iterator type

- typedef list_type::size_type **size_type**

  type of returned sizes

- typedef list_type::value_type **value_type**

  type of data in container

Public Member Functions

- **Row** ()

  Default constructor.

- **Row** (const **Row** &r)

  Copy constructor.

- **Row** (MYSQL_ROW row, const **ResultBase** ∗res, const unsigned long ∗lengths, bool te=true)

  Create a row object.

- ∼**Row** ()

  Destroy object.

- **const_reference at** (**size_type** i) const

  Get a const reference to the field given its index.

- **const_reference back** () const

  Get a reference to the last element of the vector.

- **const_iterator begin** () const

  Return a const iterator pointing to first element in the container.

- bool **empty** () const

  Returns true if container is empty.

- **const_iterator end** () const

  Return a const iterator pointing to one past the last element in the container.

- **equal_list_ba**< **FieldNames**, **Row**, quote_type0 > **equal_list** (const char ∗d=",", const char ∗e=" = ") const

  Get an "equal list" of the fields and values in this row.

- template<class Manip>
  equal_list_ba< **FieldNames**, **Row**, Manip > **equal_list** (const char ∗d, const char ∗e, Manip m) const

  Get an "equal list" of the fields and values in this row.

- **value_list_ba**< **FieldNames**, do_nothing_type0 > **field_list** (const char ∗d=",") const

  Get a list of the field names in this row.

- template<class Manip>
  value_list_ba< **FieldNames**, Manip > **field_list** (const char ∗d, Manip m) const

  Get a list of the field names in this row.

- template<class Manip>
  value_list_b< **FieldNames**, Manip > **field_list** (const char ∗d, Manip m, const std::vector< bool > &vb) const

  Get a list of the field names in this row.

- **value_list_b**< **FieldNames**, quote_type0 > **field_list** (const char ∗d, const std::vector< bool > &vb) const

  Get a list of the field names in this row.

- **value_list_b**< **FieldNames**, quote_type0 > **field_list** (const std::vector< bool > &vb) const

  Get a list of the field names in this row.

- template<class Manip>
  value_list_b< **FieldNames**, Manip > **field_list** (const char ∗d, Manip m, bool t0, bool t1=false, bool t2=false, bool t3=false, bool t4=false, bool t5=false, bool t6=false, bool t7=false, bool t8=false, bool t9=false, bool ta=false, bool tb=false, bool tc=false) const

  Get a list of the field names in this row.

- **value_list_b**< **FieldNames**, quote_type0 > **field_list** (const char ∗d, bool t0, bool t1=false, bool t2=false, bool t3=false, bool t4=false, bool t5=false, bool t6=false, bool t7=false, bool t8=false, bool t9=false, bool ta=false, bool tb=false, bool tc=false) const

  Get a list of the field names in this row.

- **value_list_b**< **FieldNames**, quote_type0 > **field_list** (bool t0, bool t1=false, bool t2=false, bool t3=false, bool t4=false, bool t5=false, bool t6=false, bool t7=false, bool t8=false, bool t9=false, bool ta=false, bool tb=false, bool tc=false) const

  Get a list of the field names in this row.

- **size_type field_num** (const char ∗name) const

  Returns a field's index given its name.

- **const_reference front** () const

  Get a reference to the first element of the vector.

- **size_type max_size** () const

  Return maximum number of elements that can be stored in container without resizing.

- **Row** & **operator=** (const **Row** &rhs)

  Assignment operator.

- **const_reference operator[ ]** (const char ∗field) const

  Get the value of a field given its name.

- **const_reference operator[ ]** (int i) const

  Get the value of a field given its index.

- **operator private_bool_type** () const

  Returns true if row object was fully initialized and has data.

- **const_reverse_iterator rbegin** () const

  Return reverse iterator pointing to first element in the container.

- **const_reverse_iterator rend** () const

  Return reverse iterator pointing to one past the last element in the container.

- **size_type size** () const

  Get the number of fields in the row.

- template<class Manip>
  value_list_ba< **Row**, Manip > **value_list** (const char ∗d=",", Manip m=quote) const

  Get a list of the values in this row.

- template<class Manip>
  value_list_b< **Row**, Manip > **value_list** (const char ∗d, const std::vector< bool > &vb, Manip m=quote) const

Get a list of the values in this row.

- **value_list_b**< **Row**, quote_type0 > **value_list** (const std::vector< bool > &vb) const

  Get a list of the values in this row.

- template<class Manip>
  value_list_b< **Row**, Manip > **value_list** (const char ∗d, Manip m, bool t0, bool t1=false, bool t2=false, bool t3=false, bool t4=false, bool t5=false, bool t6=false, bool t7=false, bool t8=false, bool t9=false, bool ta=false, bool tb=false, bool tc=false) const

  Get a list of the values in this row.

- **value_list_b**< **Row**, quote_type0 > **value_list** (const char ∗d, bool t0, bool t1=false, bool t2=false, bool t3=false, bool t4=false, bool t5=false, bool t6=false, bool t7=false, bool t8=false, bool t9=false, bool ta=false, bool tb=false, bool tc=false) const

  Get a list of the values in this row.

- **value_list_b**< **Row**, quote_type0 > **value_list** (bool t0, bool t1=false, bool t2=false, bool t3=false, bool t4=false, bool t5=false, bool t6=false, bool t7=false, bool t8=false, bool t9=false, bool ta=false, bool tb=false, bool tc=false) const

  Get a list of the values in this row.

- template<class Manip>
  value_list_b< **Row**, Manip > **value_list** (const char ∗d, Manip m, std::string s0, std::string s1="", std::string s2="", std::string s3="", std::string s4="", std::string s5="", std::string s6="", std::string s7="", std::string s8="", std::string s9="", std::string sa="", std::string sb="", std::string sc="") const

  Get a list of the values in this row.

- **value_list_b**< **Row**, quote_type0 > **value_list** (const char ∗d, std::string s0, std::string s1="", std::string s2="", std::string s3="", std::string s4="", std::string s5="", std::string s6="", std::string s7="", std::string s8="", std::string s9="", std::string sa="", std::string sb="", std::string sc="") const

  Get a list of the values in this row.

- **value_list_b**< **Row**, quote_type0 > **value_list** (std::string s0, std::string s1="", std::string s2="", std::string s3="", std::string s4="", std::string s5="", std::string s6="", std::string s7="", std::string s8="", std::string s9="", std::string sa="", std::string sb="", std::string sc="") const

  Get a list of the values in this row.

5.57.1 Detailed Description

Manages rows from a result set.

This class is like an extended version of a `const std::vector` of **mysqlpp::String** (p. 211). It adds stuff for populating the vector. As for why it's `const`, what would it mean to modify a **Row** (p. 166)? If we ever did support such semantics, it should probably actually modify the database. We can't do that if we just derive from std::vector.

Not that we could derive from std::vector even if we wanted to: `vector::operator[](size_type)` would interfere with our `operator`[](const `char*`). We can avoid this only by maintaining our own public inteface independent of that of `vector`.

5.57.2 Member Typedef Documentation

5.57.2.1 typedef std::vector<String> mysqlpp::Row::list_type

type of our internal data list

This is public because all other typedefs we have for mirroring std::vector's public interface depend on it.

5.57.2.2 typedef const_iterator mysqlpp::Row::iterator

iterator type

Note that this is just an alias for the const iterator. **Row** (p. 166) is immutable, but people are in the habit of saying 'iterator' even when they don't intend to use the iterator to modify the container, so we provide this as a convenience.

5.57.2.3 typedef const_reference mysqlpp::Row::reference

reference type

See also:

iterator (p. 171) for justification for this **const_reference** (p. 166) alias

5.57.2.4 typedef const_reverse_iterator mysqlpp::Row::reverse_iterator

mutable reverse iterator type

See also:

iterator (p. 171) for justification for this **const_reverse_iterator** (p. 166) alias

5.57.3 Constructor & Destructor Documentation

5.57.3.1 mysqlpp::Row::Row (MYSQL_ROW *row*, const ResultBase ∗ *res*, const unsigned long ∗ *lengths*, bool *te* = `true`)

Create a row object.

Parameters:

row MySQL C API row data

res result set that the row comes from

lengths length of each item in row

te if true, throw exceptions on errors

References mysqlpp::ResultBase::field_names(), mysqlpp::ResultBase::field_type(), mysqlpp::ResultBase::num_fields(), and size().

5.57.4 Member Function Documentation

5.57.4.1 const_reference mysqlpp::Row::at (size_type *i*) const `[inline]`

Get a const reference to the field given its index.

If the index value is bad, the underlying std::vector is supposed to throw an exception, according to the Standard.

Referenced by operator[ ]().

5.57.4.2 equal_list_ba< FieldNames, Row, quote_type0 > mysqlpp::Row::equal_list (const char ∗ *d* = `","`, const char ∗ *e* = `" = "`) const

Get an "equal list" of the fields and values in this row.

When inserted into a C++ stream, the delimiter 'd' will be used between the items, " = " is the relationship operator, and items will be quoted and escaped.

5.57.4.3 template<class Manip> equal_list_ba< FieldNames, Row, Manip > mysqlpp::Row::equal_list (const char ∗ *d*, const char ∗ *e*, Manip *m*) const `[inline]`

Get an "equal list" of the fields and values in this row.

This method's parameters govern how the returned list will behave when you insert it into a C++ stream:

Parameters:

d delimiter to use between items

e the operator to use between elements

m the manipulator to use for each element

For example, if d is ",", e is " = ", and m is the quote manipulator, then the field and value lists (a, b) (c, d'e) will yield an equal list that gives the following when inserted into a C++ stream:

```
'a' = 'c', 'b' = 'd''e'
```

Notice how the single quote was 'escaped' in the SQL way to avoid a syntax error.

5.57.4.4 value_list_ba< FieldNames, do_nothing_type0 > mysqlpp::Row::field_list (const char ∗ *d* = ", ") const

Get a list of the field names in this row.

When inserted into a C++ stream, the delimiter 'd' will be used between the items, and no manipulator will be used on the items.

5.57.4.5 template<class Manip> value_list_ba< FieldNames, Manip > mysqlpp::Row::field_list (const char ∗ *d*, Manip *m*) const `[inline]`

Get a list of the field names in this row.

Parameters:

d delimiter to place between the items when the list is inserted into a C++ stream

m manipulator to use before each item when the list is inserted into a C++ stream

5.57.4.6 template<class Manip> value_list_b< FieldNames, Manip > mysqlpp::Row::field_list (const char ∗ *d*, Manip *m*, const std::vector< bool > & *vb*) const `[inline]`

Get a list of the field names in this row.

Parameters:

d delimiter to place between the items when the list is inserted into a C++ stream

m manipulator to use before each item when the list is inserted into a C++ stream

vb for each true item in this list, add that field name to the returned list; ignore the others

5.57.4.7 value_list_b< FieldNames, quote_type0 > mysqlpp::Row::field_list (const char * *d*, const std::vector< bool > & *vb*) const

Get a list of the field names in this row.

Parameters:

d delimiter to place between the items when the list is inserted into a C++ stream

vb for each true item in this list, add that field name to the returned list; ignore the others

Field (p. 87) names will be quoted and escaped when inserted into a C++ stream.

5.57.4.8 value_list_b< FieldNames, quote_type0 > mysqlpp::Row::field_list (const std::vector< bool > & *vb*) const

Get a list of the field names in this row.

Parameters:

vb for each true item in this list, add that field name to the returned list; ignore the others

Field (p. 87) names will be quoted and escaped when inserted into a C++ stream, and a comma will be placed between them as a delimiter.

5.57.4.9 template<class Manip> value_list_b< FieldNames, Manip > mysqlpp::Row::field_list (const char * *d*, Manip *m*, bool *t0*, bool *t1* = `false`, bool *t2* = `false`, bool *t3* = `false`, bool *t4* = `false`, bool *t5* = `false`, bool *t6* = `false`, bool *t7* = `false`, bool *t8* = `false`, bool *t9* = `false`, bool *ta* = `false`, bool *tb* = `false`, bool *tc* = `false`) const `[inline]`

Get a list of the field names in this row.

For each true parameter, the field name in that position within the row is added to the returned list. When the list is inserted into a C++ stream, the delimiter 'd' will be placed between the items as a delimiter, and the manipulator 'm' used before each item.

5.57.4.10 value_list_b< FieldNames, quote_type0 > mysqlpp::Row::field_list (const char * *d*, bool *t0*, bool *t1* = `false`, bool *t2* = `false`, bool *t3* = `false`, bool *t4* = `false`, bool *t5* = `false`, bool *t6* = `false`, bool *t7* = `false`, bool *t8* = `false`, bool *t9* = `false`, bool *ta* = `false`, bool *tb* = `false`, bool *tc* = `false`) const

Get a list of the field names in this row.

For each true parameter, the field name in that position within the row is added to the returned list. When the list is inserted into a C++ stream, the delimiter 'd' will be placed between the items as a delimiter, and the items will be quoted and escaped.

5.57.4.11 value_list_b< FieldNames, quote_type0 > mysqlpp::Row::field_list (bool *t0*, bool *t1* = `false`, bool *t2* = `false`, bool *t3* = `false`, bool *t4* = `false`, bool *t5* = `false`, bool *t6* = `false`, bool *t7* = `false`, bool *t8* = `false`, bool *t9* = `false`, bool *ta* = `false`, bool *tb* = `false`, bool *tc* = `false`) const

Get a list of the field names in this row.

For each true parameter, the field name in that position within the row is added to the returned list. When the list is inserted into a C++ stream, a comma will be placed between the items as a delimiter, and the items will be quoted and escaped.

5.57.4.12]

const **Row::value_type** & mysqlpp::Row::operator[ ] (const char ∗ *field*) const

Get the value of a field given its name.

If the field does not exist in this row, we throw a **BadFieldName** (p. 18) exception.

This operator is fairly inefficient. operator[ ](int) is faster.

References at(), empty(), field_num(), size(), and mysqlpp::OptionalExceptions::throw_exceptions().

5.57.4.13]

const_reference mysqlpp::Row::operator[ ] (int *i*) const `[inline]`

Get the value of a field given its index.

This function is just syntactic sugar, wrapping the **at()** (p. 172) method.

It's **critical** that the parameter type be `int`, not `size_type`, because it will interfere with the `const char*` overload otherwise. row[0] is ambiguous when there isn't an int overload.

5.57.4.14 mysqlpp::Row::operator private_bool_type () const `[inline]`

Returns true if row object was fully initialized and has data.

This operator lets you use **Row** (p. 166) in bool context, which lets you do things like tell when you've run off the end of a "use" query's result set:

```
Query q("....");
if (UseQueryResult res = q.use()) {
    // Can use 'res', query succeeded
    while (Row row = res.fetch_row()) {
        // Retreived another row in the result set, can use 'row'
    }
}
```

5.57.4.15 template<class Manip> value_list_ba<Row, Manip> mysqlpp::Row::value_list (const char ∗ *d* = ", ", Manip *m* = `quote`) const `[inline]`

Get a list of the values in this row.

When inserted into a C++ stream, the delimiter 'd' will be used between the items, and the quoting and escaping rules will be set by the manipulator 'm' you choose.

Parameters:

d delimiter to use between values

m manipulator to use when inserting values into a stream

5.57.4.16 template<class Manip> value_list_b<Row, Manip> mysqlpp::Row::value_list (const char ∗ *d*, const std::vector< bool > & *vb*, Manip *m* = `quote`) const `[inline]`

Get a list of the values in this row.

Parameters:

d delimiter to use between values

vb for each true item in this list, add that value to the returned list; ignore the others

m manipulator to use when inserting values into a stream

5.57.4.17 value_list_b<Row, quote_type0> mysqlpp::Row::value_list (const std::vector< bool > & *vb*) const `[inline]`

Get a list of the values in this row.

Parameters:

vb for each true item in this list, add that value to the returned list; ignore the others

Items will be quoted and escaped when inserted into a C++ stream, and a comma will be used as a delimiter between the items.

5.57.4.18 template<class Manip> value_list_b<Row, Manip> mysqlpp::Row::value_list (const char ∗ *d*, Manip *m*, bool *t0*, bool *t1* = `false`, bool *t2* = `false`, bool *t3* = `false`, bool *t4* = `false`, bool *t5* = `false`, bool *t6* = `false`, bool *t7* = `false`, bool *t8* = `false`, bool *t9* = `false`, bool *ta* = `false`, bool *tb* = `false`, bool *tc* = `false`) const `[inline]`

Get a list of the values in this row.

For each true parameter, the value in that position within the row is added to the returned list. When the list is inserted into a C++ stream, the delimiter 'd' will be placed between the items, and the manipulator 'm' used before each item.

5.57.4.19 value_list_b<Row, quote_type0> mysqlpp::Row::value_list (const char ∗ *d*, bool *t0*, bool *t1* = `false`, bool *t2* = `false`, bool *t3* = `false`, bool *t4* = `false`, bool *t5* = `false`, bool *t6* = `false`, bool *t7* = `false`, bool *t8* = `false`, bool *t9* = `false`, bool *ta* = `false`, bool *tb* = `false`, bool *tc* = `false`) const `[inline]`

Get a list of the values in this row.

For each true parameter, the value in that position within the row is added to the returned list. When the list is inserted into a C++ stream, the delimiter 'd' will be placed between the items, and items will be quoted and escaped.

5.57.4.20 value_list_b<Row, quote_type0> mysqlpp::Row::value_list (bool *t0*, bool *t1* = `false`, bool *t2* = `false`, bool *t3* = `false`, bool *t4* = `false`, bool *t5* = `false`, bool *t6* = `false`, bool *t7* = `false`, bool *t8* = `false`, bool *t9* = `false`, bool *ta* = `false`, bool *tb* = `false`, bool *tc* = `false`) const `[inline]`

Get a list of the values in this row.

For each true parameter, the value in that position within the row is added to the returned list. When the list is inserted into a C++ stream, the a comma will be placed between the items, as a delimiter, and items will be quoted and escaped.

5.57.4.21 template<class Manip> value_list_b<Row, Manip> mysqlpp::Row::value_list (const char ∗ *d*, Manip *m*, std::string *s0*, std::string *s1* = "", std::string *s2* = "", std::string *s3* = "", std::string *s4* = "", std::string *s5* = "", std::string *s6* = "", std::string *s7* = "", std::string *s8* = "", std::string *s9* = "", std::string *sa* = "", std::string *sb* = "", std::string *sc* = "") const `[inline]`

Get a list of the values in this row.

The 's' parameters name the fields that will be added to the returned list. When inserted into a C++ stream, the delimiter 'd' will be placed between the items, and the manipulator 'm' will be inserted before each item.

5.57.4.22 value_list_b<Row, quote_type0> mysqlpp::Row::value_list (const char ∗ *d*, std::string *s0*, std::string *s1* = "", std::string *s2* = "", std::string *s3* = "", std::string *s4* = "", std::string *s5* = "", std::string *s6* = "", std::string *s7* = "", std::string *s8* = "", std::string *s9* = "", std::string *sa* = "", std::string *sb* = "", std::string *sc* = "") const `[inline]`

Get a list of the values in this row.

The 's' parameters name the fields that will be added to the returned list. When inserted into a C++ stream, the delimiter 'd' will be placed between the items, and items will be quoted and escaped.

5.57.4.23 value_list_b<Row, quote_type0> mysqlpp::Row::value_list (std::string *s0*, std::string *s1* = "", std::string *s2* = "", std::string *s3* = "", std::string *s4* = "", std::string *s5* = "", std::string *s6* = "", std::string *s7* = "", std::string *s8* = "", std::string *s9* = "", std::string *sa* = "", std::string *sb* = "", std::string *sc* = "") const `[inline]`

Get a list of the values in this row.

The 's' parameters name the fields that will be added to the returned list. When inserted into a C++ stream, a comma will be placed between the items as a delimiter, and items will be quoted and escaped.

The documentation for this class was generated from the following files:

- **row.h**
- row.cpp

5.58 mysqlpp::ScopedLock Class Reference

Wrapper around **BeecryptMutex** (p. 26) to add scope-bound locking and unlocking.

`#include <beemutex.h>`

Collaboration diagram for mysqlpp::ScopedLock:

Public Member Functions

- **ScopedLock** (**BeecryptMutex** &mutex)

  Lock the mutex.

- ∼**ScopedLock** ()

  Unlock the mutex.

5.58.1 Detailed Description

Wrapper around **BeecryptMutex** (p. 26) to add scope-bound locking and unlocking.

This allows code to lock a mutex and ensure it will unlock on exit from the enclosing scope even in the face of exceptions. This is separate from **BeecryptMutex** (p. 26) because we don't want to make this behavior mandatory.

The documentation for this class was generated from the following file:

- **beemutex.h**

5.59 mysqlpp::SecureAuthOption Class Reference

Enforce use of secure authentication, refusing connection if not available.

`#include <options.h>`

Inheritance diagram for mysqlpp::SecureAuthOption:

Collaboration diagram for mysqlpp::SecureAuthOption:

5.59.1 Detailed Description

Enforce use of secure authentication, refusing connection if not available.

The documentation for this class was generated from the following file:

- **options.h**

5.60 mysqlpp::SelfTestFailed Class Reference

Used within MySQL++'s test harness only.

`#include <exceptions.h>`

Inheritance diagram for mysqlpp::SelfTestFailed:

Collaboration diagram for mysqlpp::SelfTestFailed:

Public Member Functions

- **SelfTestFailed** (const std::string &w)
  Create exception object.

5.60.1 Detailed Description

Used within MySQL++'s test harness only.

The documentation for this class was generated from the following file:

- **exceptions.h**

5.61 mysqlpp::Set< Container > Class Template Reference

A special std::set derivative for holding MySQL data sets.

`#include <myset.h>`

Public Member Functions

- **Set** ()

  Default constructor.

- **Set** (const char ∗str)

  Create object from a comma-separated list of values.

- **Set** (const std::string &str)

  Create object from a comma-separated list of values.

- **Set** (const **String** &str)

  Create object from a comma-separated list of values.

- **operator std::string** () const

  Convert this set's data to a string containing comma-separated items.

- std::string **str** () const

  Return our value in std::string form.

5.61.1 Detailed Description

template<class Container = std::set<std::string>> class mysqlpp::Set< Container >

A special std::set derivative for holding MySQL data sets.

The documentation for this class was generated from the following file:

- **myset.h**

5.62 mysqlpp::SetCharsetDirOption Class Reference

Give path to charset definition files.

`#include <options.h>`

Inheritance diagram for mysqlpp::SetCharsetDirOption:

Collaboration diagram for mysqlpp::SetCharsetDirOption:

5.62.1 Detailed Description

Give path to charset definition files.

The documentation for this class was generated from the following file:

- **options.h**

5.63 mysqlpp::SetCharsetNameOption Class Reference

Give name of default charset.

`#include <options.h>`

Inheritance diagram for mysqlpp::SetCharsetNameOption:

Collaboration diagram for mysqlpp::SetCharsetNameOption:

5.63.1 Detailed Description

Give name of default charset.

The documentation for this class was generated from the following file:

- **options.h**

5.64 mysqlpp::SetClientIpOption Class Reference

Fake client IP address when connecting to embedded server.

`#include <options.h>`

Inheritance diagram for mysqlpp::SetClientIpOption:

Collaboration diagram for mysqlpp::SetClientIpOption:

5.64.1 Detailed Description

Fake client IP address when connecting to embedded server.

The documentation for this class was generated from the following file:

- **options.h**

5.65 mysqlpp::SharedMemoryBaseNameOption Class Reference

Set (p. 182) name of shmem segment for IPC.

`#include <options.h>`

Inheritance diagram for mysqlpp::SharedMemoryBaseNameOption:

Collaboration diagram for mysqlpp::SharedMemoryBaseNameOption:

5.65.1 Detailed Description

Set (p. 182) name of shmem segment for IPC.

The documentation for this class was generated from the following file:

- **options.h**

5.66 mysqlpp::SimpleResult Class Reference

Holds information about the result of queries that don't return rows.

`#include <result.h>`

Collaboration diagram for mysqlpp::SimpleResult:

Public Member Functions

- **SimpleResult** ()

  Default ctor.

- **SimpleResult** (bool copacetic, ulonglong insert_id, ulonglong rows, const std::string &info)

  Initialize object.

- **operator private_bool_type** () const

  Test whether the query that created this result succeeded.

- ulonglong **insert_id** () const

  Get the last value used for an AUTO_INCREMENT field.

- ulonglong **rows** () const

  Get the number of rows affected by the query.

- const char ∗ **info** () const

  Get any additional information about the query returned by the server.

5.66.1 Detailed Description

Holds information about the result of queries that don't return rows.

5.66.2 Member Function Documentation

5.66.2.1 mysqlpp::SimpleResult::operator private_bool_type () const `[inline]`

Test whether the query that created this result succeeded.

If you test this object in bool context and it's false, it's a signal that the query this was created from failed in some way. Call **Query::error()** (p. 134) or **Query::errnum()** (p. 134) to find out what exactly happened.

The documentation for this class was generated from the following file:

- **result.h**

5.67 mysqlpp::SQLBuffer Class Reference

Holds SQL data in string form plus type information for use in converting the string to compatible C++ data types.

```
#include <sql_buffer.h>
```

Collaboration diagram for mysqlpp::SQLBuffer:

Public Types

- typedef size_t **size_type**

  Type of length values.

Public Member Functions

- **SQLBuffer** (const char ∗data, **size_type** length, **mysql_type_info** type, bool is_null)

  Initialize object as a copy of a raw data buffer.

- **SQLBuffer** (const std::string &s, **mysql_type_info** type, bool is_null)

  Initialize object as a copy of a C++ string object.

- ∼**SQLBuffer** ()

  Destructor.

- **SQLBuffer** & **assign** (const char ∗data, **size_type** length, **mysql_type_info** type=**mysql_type_info::string_type**, bool is_null=false)

  Replace contents of buffer with copy of given C string.

- **SQLBuffer** & **assign** (const std::string &s, **mysql_type_info** type=**mysql_-type_info::string_type**, bool is_null=false)

  Replace contents of buffer with copy of given C++ string.

- const char ∗ **data** () const

Return pointer to raw data buffer.

- bool **escape_q** () const

  Returns true if we were initialized with a data type that must be escaped when used in a SQL query.

- **size_type length** () const

  Return number of bytes in data buffer.

- bool **is_string** ()

  Returns true if type of buffer's contents is string.

- bool **is_null** () const

  Return true if buffer's contents represent a SQL null.

- bool **quote_q** () const

  Returns true if we were initialized with a data type that must be quoted when used in a SQL query.

- void **set_null** ()

  Sets the internal SQL null flag.

- const **mysql_type_info** & **type** () const

  Return the SQL type of the data held in the buffer.

5.67.1 Detailed Description

Holds SQL data in string form plus type information for use in converting the string to compatible C++ data types.

5.67.2 Constructor & Destructor Documentation

5.67.2.1 mysqlpp::SQLBuffer::SQLBuffer (const char ∗ *data*, size_type *length*, mysql_type_info *type*, bool *is_null*) `[inline]`

Initialize object as a copy of a raw data buffer.

Copies the string into a new buffer one byte longer than the length value given, using that to hold a C string null terminator, just for safety. The length value we keep does not include this extra byte, allowing this same mechanism to work for both C strings and binary data.

5.67.3 Member Function Documentation

5.67.3.1 size_type mysqlpp::SQLBuffer::length () const `[inline]`

Return number of bytes in data buffer.

Count does not include the trailing null we tack on to our copy of the buffer for ease of use in C string contexts. We do this because we can be holding binary data just as easily as a C string.

5.67.3.2 bool mysqlpp::SQLBuffer::is_null () const `[inline]`

Return true if buffer's contents represent a SQL null.

The buffer's actual content will probably be "NULL" or something like it, but in the SQL data type system, a SQL null is distinct from a plain string with value "NULL".

The documentation for this class was generated from the following files:

- **sql_buffer.h**
- sql_buffer.cpp

5.68 mysqlpp::SQLParseElement Struct Reference

Used within **Query** (p. 126) to hold elements for parameterized queries.

`#include <qparms.h>`

Collaboration diagram for mysqlpp::SQLParseElement:

Public Member Functions

- **SQLParseElement** (std::string b, char o, signed char n)

    Create object.

Public Attributes

- std::string **before**

    string inserted before the parameter

- char **option**

    the parameter option, or blank if none

- signed char **num**

    the parameter position to use

5.68.1 Detailed Description

Used within **Query** (p. 126) to hold elements for parameterized queries.

Each element has three parts:

The concept behind the `before` variable needs a little explaining. When a template query is parsed, each parameter is parsed into one of these **SQLParseElement** (p. 192) objects, but the non-parameter parts of the template also have to be stored somewhere. MySQL++ chooses to attach the text leading up to a parameter to that parameter. So, the `before` string is simply the text copied literally into the finished query before we insert a value for the parameter.

The `option` character is currently one of ’q’, ’Q’, ’r’, ’R’ or ’ ’. See the "Template Queries" chapter in the user manual for details.

The position value (`num`) allows a template query to have its parameters in a different order than in the **Query** (p. 126) method call. An example of how this can be helpful is in the "Template Queries" chapter of the user manual.

5.68.2 Constructor & Destructor Documentation

5.68.2.1 mysqlpp::SQLParseElement::SQLParseElement (std::string *b*, char *o*, signed char *n*) `[inline]`

Create object.

Parameters:

b the ’before’ value

o the ’option’ value

n the ’num’ value

The documentation for this struct was generated from the following file:

- **qparms.h**

5.69 mysqlpp::SQLQueryParms Class Reference

This class holds the parameter values for filling template queries.

`#include <qparms.h>`

Collaboration diagram for mysqlpp::SQLQueryParms:

Public Types

- typedef const **SQLTypeAdapter** & **sta**

  Abbreviation so some of the declarations below don't span many lines.

Public Member Functions

- **SQLQueryParms** ()

  Default constructor.

- **SQLQueryParms** (**Query** ∗p)

  Create object.

- bool **bound** ()

  Returns true if we are bound to a query object.

- void **clear** ()

  Clears the list.

- size_t **escape_string** (std::string ∗ps, const char ∗original=0, size_t length=0) const

Indirect access to ***Query::escape_string()*** *(p. 133).*

- size_t **escape_string** (char ∗escaped, const char ∗original, size_t length) const

  Indirect access to ***Query::escape_string()*** *(p. 133).*

- **SQLTypeAdapter** & **operator[ ]** (size_type n)

  Access element number n.

- const **SQLTypeAdapter** & **operator[ ]** (size_type n) const

  Access element number n.

- **SQLTypeAdapter** & **operator[ ]** (const char ∗str)

  Access the value of the element with a key of str.

- const **SQLTypeAdapter** & **operator[ ]** (const char ∗str) const

  Access the value of the element with a key of str.

- **SQLQueryParms** & **operator**<< (const **SQLTypeAdapter** &str)

  Adds an element to the list.

- **SQLQueryParms** & **operator+=** (const **SQLTypeAdapter** &str)

  Adds an element to the list.

- **SQLQueryParms operator+** (const **SQLQueryParms** &other) const

  Build a composite of two parameter lists.

- void **set** (**sta** a, **sta** b, **sta** c, **sta** d, **sta** e, **sta** f, **sta** g, **sta** h, **sta** i, **sta** j, **sta** k, **sta** l)

  Set *(p. 182) the template query parameters.*

Friends

- class **Query**

5.69.1 Detailed Description

This class holds the parameter values for filling template queries.

5.69.2 Constructor & Destructor Documentation

5.69.2.1 mysqlpp::SQLQueryParms::SQLQueryParms (Query ∗ *p*) `[inline]`

Create object.

Parameters:

p pointer to the query object these parameters are tied to

5.69.3 Member Function Documentation

5.69.3.1 bool mysqlpp::SQLQueryParms::bound () `[inline]`

Returns true if we are bound to a query object.

Basically, this tells you which of the two ctors were called.

5.69.3.2 size_t mysqlpp::SQLQueryParms::escape_string (char ∗ *escaped*, const char ∗ *original*, size_t *length*) const

Indirect access to **Query::escape_string**() (p. 133).

See also:

escape_string(std::string∗, const char∗, size_t)
Query::escape_string(const char∗, const char∗, size_t)

References mysqlpp::Query::escape_string().

5.69.3.3 SQLQueryParms mysqlpp::SQLQueryParms::operator+ (const SQLQueryParms & *other*) const

Build a composite of two parameter lists.

If this list is (a, b) and `other` is (c, d, e, f, g), then the returned list will be (a, b, e, f, g). That is, all of this list's parameters are in the returned list, plus any from the other list that are in positions beyond what exist in this list.

If the two lists are the same length or this list is longer than the `other` list, a copy of this list is returned.

5.69.3.4 void mysqlpp::SQLQueryParms::set (sta *a*, sta *b*, sta *c*, sta *d*, sta *e*, sta *f*, sta *g*, sta *h*, sta *i*, sta *j*, sta *k*, sta *l*) `[inline]`

Set (p. 182) the template query parameters.

Sets parameter 0 to a, parameter 1 to b, etc. There are overloaded versions of this function that take anywhere from one to a dozen parameters.

The documentation for this class was generated from the following files:

- **qparms.h**
- qparms.cpp

5.70 mysqlpp::SQLTypeAdapter Class Reference

Converts many different data types to strings suitable for use in SQL queries.

`#include <stadapter.h>`

Public Types

- typedef size_t **size_type**

  size of length values

Public Member Functions

- **SQLTypeAdapter** ()

  Default constructor; empty string.

- **SQLTypeAdapter** (const **SQLTypeAdapter** &other)

  Copy ctor.

- **SQLTypeAdapter** (const **String** &str, bool processed=false)

  Create a copy of a MySQL++ string.

- **SQLTypeAdapter** (const std::string &str, bool processed=false)

  Create a copy of a C++ string.

- **SQLTypeAdapter** (const char ∗str, bool processed=false)

  Create a copy of a null-terminated C string.

- **SQLTypeAdapter** (const char ∗str, int len, bool processed=false)

  Create a copy of an arbitrary block of data.

- **SQLTypeAdapter** (char c)

  Create a single-character string.

- **SQLTypeAdapter** (sql_tinyint i)

  Create a string representation of SQL `TINYINT`.

- **SQLTypeAdapter** (sql_tinyint_unsigned i)

  Create a string representation of SQL `TINYINT UNSIGNED`.

- **SQLTypeAdapter** (short i)

Create a string representation of a `short int` *value.*

- **SQLTypeAdapter** (unsigned short i)

  Create a string representation of an `unsigned short int` *value.*

- **SQLTypeAdapter** (int i)

  Create a string representation of an `int` *value.*

- **SQLTypeAdapter** (unsigned i)

  Create a string representation of an `unsigned int` *value.*

- **SQLTypeAdapter** (long i)

  Create a string representation of a `long int` *value.*

- **SQLTypeAdapter** (unsigned long i)

  Create a string representation of an `unsigned long int` *value.*

- **SQLTypeAdapter** (longlong i)

  Create a string representation of a `longlong` *value.*

- **SQLTypeAdapter** (ulonglong i)

  Create a string representation of an `unsigned longlong` *value.*

- **SQLTypeAdapter** (float i)

  Create a string representation of a `float` *value.*

- **SQLTypeAdapter** (double i)

  Create a string representation of a `double` *value.*

- **SQLTypeAdapter** (const **Date** &d)

  Create a SQL string representation of a date.

- **SQLTypeAdapter** (const **DateTime** &dt)

  Create a SQL string representation of a date and time.

- **SQLTypeAdapter** (const **Time** &t)

  Create a SQL string representation of a time.

- **SQLTypeAdapter** (const **null_type** &i)

  Create object representing SQL NULL.

- **SQLTypeAdapter** & **operator=** (const **SQLTypeAdapter** &rhs)

Standard assignment operator.

- **SQLTypeAdapter** & **operator=** (const **null_type** &n)

  Replace contents of object with a SQL null.

- **operator const char** ∗ () const

  Returns a const char pointer to the object's raw data.

- **SQLTypeAdapter** & **assign** (const **SQLTypeAdapter** &sta)

  Copies another SQLTypeAdapter's data buffer into this object.

- **SQLTypeAdapter** & **assign** (const char ∗pc, int len=-1)

  Copies a C string or a raw buffer into this object.

- **SQLTypeAdapter** & **assign** (const **null_type** &n)

  Replaces contents of object with a SQL null.

- char **at** (**size_type** i) const throw (std::out_of_range)

  Returns the character at a given position within the string buffer.

- int **compare** (const **SQLTypeAdapter** &other) const

  Compare the internal buffer to the given string.

- int **compare** (const std::string &other) const

  Compare the internal buffer to the given string.

- int **compare** (**size_type** pos, **size_type** num, std::string &other) const

  Compare the internal buffer to the given string.

- int **compare** (const char ∗other) const

  Compare the internal buffer to the given string.

- int **compare** (**size_type** pos, **size_type** num, const char ∗other) const

  Compare the internal buffer to the given string.

- const char ∗ **data** () const

  Return pointer to raw data buffer.

- bool **escape_q** () const

  Returns true if we were initialized with a data type that must be escaped when used in a SQL query.

- bool **is_processed** () const

Returns true if the internal 'processed' flag is set.

- **size_type length** () const

  Return number of bytes in data buffer.

- **size_type size** () const

  alias for ***length()*** *(p. 201)*

- bool **quote_q** () const

  Returns true if we were initialized with a data type that must be quoted when used in a SQL query.

- int **type_id** () const

  Returns the type ID of the buffer's data.

- void **set_processed** ()

  Turns on the internal 'is_processed_' flag.

5.70.1 Detailed Description

Converts many different data types to strings suitable for use in SQL queries.

This class provides implicit conversion between many C++ types and SQL-formatted string representations of that data without losing important type information. This class is not for direct use outside MySQL++ itself. It exists for those interfaces in MySQL++ that need to accept a value of any reasonable data type which it will use in building a query string.

One major use for this is in the **Query** (p. 126) class interfaces for building template queries: they have to be generic with respect to argument type, but because we know we want the data in some kind of string form eventually, we don't need to templatize it. The interface can just use **SQLTypeAdapter** (p. 198), which lets callers pass any reasonable data type. The adapter converts the passed value implicitly.

The other major use for this type is the quoting and escaping logic in Query's stream interface: rather than overload the << operators and the manipulators for every single type we know the rules for *a priori*, we just specialize the manipulators for **SQLTypeAdapter** (p. 198). The conversion to **SQLTypeAdapter** (p. 198) stringizes the data, which we needed anyway for stream insertion, and holds enough type information so that the manipulator can decide whether to do automatic quoting and/or escaping.

5.70.2 Constructor & Destructor Documentation

5.70.2.1 mysqlpp::SQLTypeAdapter::SQLTypeAdapter (const SQLTypeAdapter & *other*)

Copy ctor.

Parameters:

other the other **SQLTypeAdapter** (p. 198) object

This ctor only copies the pointer to the other SQLTypeAdapter's data buffer and increments its reference counter. If you need a deep copy, use one of the ctors that takes a string.

5.70.2.2 mysqlpp::SQLTypeAdapter::SQLTypeAdapter (const String & *str*, bool *processed* = `false`)

Create a copy of a MySQL++ string.

This does reference-counted buffer sharing with the other object. If you need a deep copy, pass the result of either **String::c_str()** (p. 212) or **String::conv()** (p. 212) instead, which will call one of the other string ctors.

5.70.2.3 mysqlpp::SQLTypeAdapter::SQLTypeAdapter (char *c*)

Create a single-character string.

If you mean for `c` to be treated as a small integer, you should be using **mysqlpp::tiny_int** (p. 232) instead. It avoids the confusion in C++ between integer and character. See the documentation for **tiny_int.h** (p. 297) for details.

5.70.3 Member Function Documentation

5.70.3.1 SQLTypeAdapter & mysqlpp::SQLTypeAdapter::operator= (const SQLTypeAdapter & *rhs*)

Standard assignment operator.

See also:

assign(const SQLTypeAdapter&) (p. 203) for details

References assign().

5.70.3.2 SQLTypeAdapter & mysqlpp::SQLTypeAdapter::operator= (const null_type & *n*)

Replace contents of object with a SQL null.

See also:

assign(const null_type&) (p. 204) for details

References assign().

5.70.3.3 SQLTypeAdapter & mysqlpp::SQLTypeAdapter::assign (const SQLTypeAdapter & *sta*)

Copies another SQLTypeAdapter's data buffer into this object.

Parameters:

sta Other object to copy

Return values:

****this*** Detaches this object from its internal buffer and attaches itself to the other object's buffer, with reference counting on each side. If you need a deep copy, call one of the **assign()** (p. 203) overloads taking a C or C++ string instead.

References buffer_.

Referenced by operator=().

5.70.3.4 SQLTypeAdapter & mysqlpp::SQLTypeAdapter::assign (const char * *pc*, int *len* = `-1`)

Copies a C string or a raw buffer into this object.

Parameters:

pc Pointer to char buffer to copy

len Number of characters to copy; default tells function to use the return value of strlen() instead.

Return values:

****this*** If you give the len parameter, this function will treat pc as a pointer to an array of char, not as a C string. It only treats null characters as special when you leave len at its default.

References mysqlpp::mysql_type_info::string_type.

5.70.3.5 SQLTypeAdapter & mysqlpp::SQLTypeAdapter::assign (const null_type & *n*)

Replaces contents of object with a SQL null.

Parameters:

n typically, the MySQL++ global object mysqlpp::null

Return values:

∗***this***

5.70.3.6 char mysqlpp::SQLTypeAdapter::at (size_type *i*) const throw (std::out_of_range)

Returns the character at a given position within the string buffer.

Exceptions:

out_of_range if the internal buffer is not initialized (default ctor called, and no assignment operator subsequently) or if there are not at least i + 1 characters in the buffer

References length().

5.70.3.7 int mysqlpp::SQLTypeAdapter::compare (const SQLTypeAdapter & *other*) const

Compare the internal buffer to the given string.

Works just like string::compare(const std::string&).

References buffer_, and length().

Referenced by compare().

5.70.3.8 int mysqlpp::SQLTypeAdapter::compare (const std::string & *other*) const

Compare the internal buffer to the given string.

Works just like string::compare(const std::string&).

References compare(), and length().

5.70.3.9 int mysqlpp::SQLTypeAdapter::compare (size_type *pos*, size_type *num*, std::string & *other*) const

Compare the internal buffer to the given string.

Works just like string::compare(size_type, size_type, std::string&).

References compare().

5.70.3.10 int mysqlpp::SQLTypeAdapter::compare (const char ∗ *other*) const

Compare the internal buffer to the given string.

Works just like string::compare(const char∗).

References compare(), and length().

5.70.3.11 int mysqlpp::SQLTypeAdapter::compare (size_type *pos*, size_type *num*, const char ∗ *other*) const

Compare the internal buffer to the given string.

Works just like string::compare(size_type, size_type, const char∗).

References data().

5.70.3.12 bool mysqlpp::SQLTypeAdapter::is_processed () const `[inline]`

Returns true if the internal 'processed' flag is set.

This is an implementation detail of template queries, used to prevent repeated processing of values.

5.70.3.13 int mysqlpp::SQLTypeAdapter::type_id () const

Returns the type ID of the buffer's data.

Values from **type_info.h** (p. 299). At the moment, these are the same as the underlying MySQL C API type IDs, but it's not a good idea to count on this remaining the case.

5.70.3.14 void mysqlpp::SQLTypeAdapter::set_processed () `[inline]`

Turns on the internal 'is_processed_' flag.

This is an implementation detail of template queries, used to prevent repeated processing of values.

The documentation for this class was generated from the following files:

- **stadapter.h**
- stadapter.cpp

5.71 mysqlpp::SslOption Class Reference

Specialized option for handling SSL parameters.

`#include <options.h>`

Inheritance diagram for mysqlpp::SslOption:

Collaboration diagram for mysqlpp::SslOption:

Public Member Functions

- **SslOption** (const char ∗key=0, const char ∗cert=0, const char ∗ca=0, const char ∗capath=0, const char ∗cipher=0)

  Create a set of SSL connection option parameters.

5.71.1 Detailed Description

Specialized option for handling SSL parameters.

5.71.2 Constructor & Destructor Documentation

5.71.2.1 mysqlpp::SslOption::SslOption (const char ∗ *key* = `0`, const char ∗ *cert* = `0`, const char ∗ *ca* = `0`, const char ∗ *capath* = `0`, const char ∗ *cipher* = `0`) `[inline]`

Create a set of SSL connection option parameters.

Parameters:

key the pathname to the key file

cert the pathname to the certificate file

ca the pathname to the certificate authority file

capath directory that contains trusted SSL CA certificates in pem format.

cipher list of allowable ciphers to use This option replaces `Connection::enable_ssl()` from MySQL++ version 2. Now you can set this connection option just like any other.

The documentation for this class was generated from the following file:

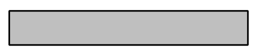
- **options.h**

5.72 mysqlpp::StoreQueryResult Class Reference

StoreQueryResult (p. 209) set type for "store" queries.

`#include <result.h>`

Inheritance diagram for mysqlpp::StoreQueryResult:



Collaboration diagram for mysqlpp::StoreQueryResult:

Public Types

- typedef std::vector< **Row** > **list_type**
  type of vector base class

Public Member Functions

- **StoreQueryResult** ()
  Default constructor.
- **StoreQueryResult** (MYSQL_RES ∗result, **DBDriver** ∗dbd, bool te=true)
  Fully initialize object.
- **StoreQueryResult** (const **StoreQueryResult** &other)

*Initialize object as a copy of another **StoreQueryResult** (p. 209) object.*

- ∼**StoreQueryResult** ()

  Destroy result set.

- list_type::size_type **num_rows** () const

  Returns the number of rows in this result set.

- **StoreQueryResult** & **operator=** (const **StoreQueryResult** &rhs)

  *Copy another **StoreQueryResult** (p. 209) object's data into this object.*

- **operator private_bool_type** () const

  Test whether the query that created this result succeeded.

5.72.1 Detailed Description

StoreQueryResult (p. 209) set type for "store" queries.

This is the obvious C++ implementation of a class to hold results from a SQL query that returns rows: a specialization of std::vector holding **Row** (p. 166) objects in memory so you get random-access semantics. MySQL++ also supports **UseQueryResult** (p. 247) which is less friendly, but has better memory performance. See the user manual for more details on the distinction and the usage patterns required.

5.72.2 Member Function Documentation

5.72.2.1 mysqlpp::StoreQueryResult::operator private_bool_type () const `[inline]`

Test whether the query that created this result succeeded.

If you test this object in bool context and it's false, it's a signal that the query this was created from failed in some way. Call **Query::error()** (p. 134) or **Query::errnum()** (p. 134) to find out what exactly happened.

The documentation for this class was generated from the following files:

- **result.h**
- result.cpp

5.73 mysqlpp::String Class Reference

A std::string work-alike that can convert itself from SQL text data formats to C++ data types.

```
#include <mystring.h>
```

Public Types

- typedef const char **value_type**

  Type of the data stored in this object, when it is not equal to SQL null.

- typedef size_t **size_type**

  Type of "size" integers.

- typedef const char ∗ **const_iterator**

  Type of iterators.

- typedef **const_iterator iterator**

  Same as const_iterator because the data cannot be changed.

Public Member Functions

- **String** ()

  Default constructor.

- **String** (const **String** &other)

  Copy ctor.

- **String** (const char ∗str, **size_type** len, **mysql_type_info** type=**mysql_type_-info::string_type**, bool is_null=false)

  Full constructor.

- **String** (const std::string &str, **mysql_type_info** type=**mysql_type_-info::string_type**, bool is_null=false)

  C++ string version of full ctor.

- **String** (const char ∗str, **mysql_type_info** type=**mysql_type_info::string_type**, bool is_null=false)

  Null-terminated C string version of full ctor.

- ∼**String** ()

  Destroy string.

- void **assign** (const char ∗str, **size_type** len, **mysql_type_info** type=**mysql_-type_info::string_type**, bool is_null=false)

  Assign raw data to this object.

- void **assign** (const std::string &str, **mysql_type_info** type=**mysql_type_-info::string_type**, bool is_null=false)

  Assign a C++ string to this object.

- void **assign** (const char ∗str, **mysql_type_info** type=**mysql_type_info::string_-type**, bool is_null=false)

  Assign a C string to this object.

- char **at** (**size_type** pos) const

  Return a character within the string.

- **const_iterator begin** () const

  Return iterator pointing to the first character of the string.

- const char ∗ **c_str** () const

  Return a const pointer to the string data.

- template<class Type>
  Type **conv** (Type) const

  Template for converting the column data to most any numeric data type.

- template<class T, class B>
  Null< T, B > **conv** (**Null**< T, B >) const

  *Overload of **conv()** (p. 212) for types wrapped with Null<>.*

- int **compare** (const **String** &other) const

  Lexically compare this string to another.

- int **compare** (const std::string &other) const

  Lexically compare this string to another.

- int **compare** (**size_type** pos, **size_type** num, std::string &other) const

  Lexically compare this string to another.

- int **compare** (const char ∗other) const

  Lexically compare this string to another.

- int **compare** (**size_type** pos, **size_type** num, const char ∗other) const

  Lexically compare this string to another.

- const char ∗ **data** () const

  Raw access to the underlying buffer, with no C string interpretation.

- **const_iterator end** () const

  Return iterator pointing to one past the last character of the string.

- bool **escape_q** () const

  Returns true if data of this type should be escaped, false otherwise.

- bool **is_null** () const

  Returns true if this object is a SQL null.

- void **it_is_null** ()

  ***Set** (*p.* 182) a flag indicating that this object is a SQL null.*

- **size_type length** () const

  Return number of bytes in the string.

- **size_type max_size** () const

  Return the maximum number of characters in the string.

- bool **quote_q** () const

  Returns true if data of this type should be quoted, false otherwise.

- **size_type size** () const

  Return number of bytes in string.

- void **strip_leading_blanks** (std::string &s) const

  Returns a copy of our internal string without leading blanks.

- void **to_string** (std::string &s) const

  Copies this object's data into a C++ string.

- **mysql_type_info type** () const

  Get this object's current MySQL type.

- **String** & **operator=** (const std::string &rhs)

  Assignment operator, from C++ string.

- **String** & **operator=** (const char ∗str)

  Assignment operator, from C string.

- **String** & **operator=** (const **String** &other)

  *Assignment operator, from other **String** (p. 211).*

- template<typename T>
  bool **operator==** (const T &rhs) const

  Equality comparison operator.

- bool **operator==** (const **mysqlpp::null_type** &) const

  Equality comparison operator.

- template<typename T>
  bool **operator!=** (const T &rhs) const

  Inequality comparison operator.

- bool **operator!=** (const **mysqlpp::null_type** &) const

  Inequality comparison operator.

- char **operator[ ]** (**size_type** pos) const

  Return a character within the string.

- **operator const char** ∗ () const

  Returns a const char pointer to the object's raw data.

- **operator signed char** () const

  Converts this object's string data to a signed char.

- **operator unsigned char** () const

  Converts this object's string data to an unsigned char.

- **operator int** () const

  Converts this object's string data to an int.

- **operator unsigned int** () const

  Converts this object's string data to an unsigned int.

- **operator short int** () const

  Converts this object's string data to a short int.

- **operator unsigned short int** () const

Converts this object's string data to an unsigned short int.

- **operator long int** () const

  Converts this object's string data to a long int.

- **operator unsigned long int** () const

  Converts this object's string data to an unsigned long int.

- **operator longlong** () const

  Converts this object's string data to the platform- specific 'longlong' type, usually a 64-bit integer.

- **operator ulonglong** () const

  Converts this object's string data to the platform- specific 'ulonglong' type, usually a 64-bit unsigned integer.

- **operator float** () const

  Converts this object's string data to a float.

- **operator double** () const

  Converts this object's string data to a double.

- **operator bool** () const

  Converts this object's string data to a bool.

- **operator Date** () const

  Converts this object's string data to a ***mysqlpp::Date*** *(*p.*53).*

- **operator DateTime** () const

  Converts this object's string data to a ***mysqlpp::DateTime*** *(*p.*57).*

- **operator Time** () const

  Converts this object's string data to a ***mysqlpp::Time*** *(*p.*228).*

- template<class T, class B>
  operator Null< **T, B** > () const

  Converts the ***String*** *(*p.*211) to a nullable data type.*

Friends

- class **SQLTypeAdapter**

5.73.1 Detailed Description

A std::string work-alike that can convert itself from SQL text data formats to C++ data types.

This class is an intermediate form for a SQL field, normally converted to a more useful native C++ type, not used directly. The only exception is in dealing with BLOB data, which stays in **String** (p. 211) form for efficiency and to avoid corrupting the data with facile conversions. Even then, it's best to use it through the typedef aliases like sql_blob in **sql_types.h** (p. 293), in case we later change this underlying representation.

String's implicit conversion operators let you can use these objects naturally:

```
String("12.86") + 2.0
```

That will give you 14.86 (approximately) as you expect, but be careful not to get tripped up by C++'s type conversion rules. If you had said this instead:

```
String("12.86") + 2
```

the result would be 14 because 2 is an integer, and C++'s type conversion rules put the **String** (p. 211) object in an integer context.

You can disable the operator overloads that allow these things by defining MYSQLPP_-NO_BINARY_OPERS.

This class also has some basic information about the type of data stored in it, to allow it to do the conversions more intelligently than a trivial implementation would allow.

5.73.2 Constructor & Destructor Documentation

5.73.2.1 mysqlpp::String::String () `[inline]`

Default constructor.

An object constructed this way is essentially useless, but sometimes you just need to construct a default object.

5.73.2.2 mysqlpp::String::String (const String & *other*) `[inline]`

Copy ctor.

Parameters:

other the other **String** (p. 211) object

This ctor only copies the pointer to the other String's data buffer and increments its reference counter. If you need a deep copy, use one of the ctors that takes a string.

5.73.2.3 mysqlpp::String::String (const char ∗ *str*, size_type *len*, mysql_type_info *type* = mysql_type_info::string_type, bool *is_null* = `false`) `[inline, explicit]`

Full constructor.

Parameters:

str the string this object represents, or 0 for SQL null

len the length of the string; embedded nulls are legal

type MySQL type information for data within str

is_null string represents a SQL null, not literal data

The resulting object will contain a copy of the string buffer. The buffer will actually be 1 byte longer than the value given for `len`, to hold a null terminator for safety. We do this because this ctor may be used for things other than null-terminated C strings. (e.g. BLOB data)

5.73.2.4 mysqlpp::String::String (const std::string & *str*, mysql_type_info *type* = mysql_type_info::string_type, bool *is_null* = `false`) `[inline, explicit]`

C++ string version of full ctor.

Parameters:

str the string this object represents, or 0 for SQL null

type MySQL type information for data within str

is_null string represents a SQL null, not literal data

The resulting object will contain a copy of the string buffer.

5.73.2.5 mysqlpp::String::String (const char ∗ *str*, mysql_type_info *type* = mysql_type_info::string_type, bool *is_null* = `false`) `[inline, explicit]`

Null-terminated C string version of full ctor.

Parameters:

str the string this object represents, or 0 for SQL null

type MySQL type information for data within str

is_null string represents a SQL null, not literal data

The resulting object will contain a copy of the string buffer.

5.73.3 Member Function Documentation

5.73.3.1 void mysqlpp::String::assign (const char ∗ *str*, size_type *len*, mysql_type_info *type* = mysql_type_info::string_type, bool *is_null* = `false`) `[inline]`

Assign raw data to this object.

This parallels the ctor with the same parameters, for when you must do a 2-step create, or when you want to reassign the data without creating a **String** (p. 211) temporary to get around the fact that **operator=()** (p. 213) can only take one parameter.

5.73.3.2 void mysqlpp::String::assign (const std::string & *str*, mysql_type_info *type* = mysql_type_info::string_type, bool *is_null* = `false`) `[inline]`

Assign a C++ string to this object.

This parallels the ctor with the same parameters, for when you must do a 2-step create, or when you want to reassign the data without creating a **String** (p. 211) temporary to get around the fact that **operator=()** (p. 213) can only take one parameter.

5.73.3.3 void mysqlpp::String::assign (const char ∗ *str*, mysql_type_info *type* = mysql_type_info::string_type, bool *is_null* = `false`) `[inline]`

Assign a C string to this object.

This parallels the ctor with the same parameters, for when you must do a 2-step create, or when you want to reassign the data without creating a **String** (p. 211) temporary to get around the fact that **operator=()** (p. 213) can only take one parameter.

5.73.3.4 char mysqlpp::String::at (size_type *pos*) const

Return a character within the string.

Unlike **`operator[]()`** `(p. 222)`, this function throws an `std::out_of_range` exception if the index isn't within range.

References size().

5.73.3.5 template<class T, class B> Null<T, B> mysqlpp::String::conv (Null< T, B >) const `[inline]`

Overload of **conv()** (p. 212) for types wrapped with Null<>.

If the **String** (p. 211) object was initialized with some string we recognize as a SQL null, we just return a copy of the global 'null' object converted to the requested type. Otherwise, we return the String's value wrapped in the Null<> template.

5.73.3.6 int mysqlpp::String::compare (const String & *other*) const

Lexically compare this string to another.

Parameters:

other string to compare against this one

See also:

compare(size_type, size_type, const char∗)

References buffer_, and length().

Referenced by compare().

5.73.3.7 int mysqlpp::String::compare (const std::string & *other*) const

Lexically compare this string to another.

Parameters:

other string to compare against this one

See also:

compare(size_type, size_type, const char∗)

References compare(), and length().

5.73.3.8 int mysqlpp::String::compare (size_type *pos*, size_type *num*, std::string & *other*) const

Lexically compare this string to another.

Parameters:

pos position within this string to begin comparison

num maximum number of characters within this string to use in comparison

other string to compare against this one

See also:

compare(size_type, size_type, const char∗)

References compare().

5.73.3.9 int mysqlpp::String::compare (const char ∗ *other*) const

Lexically compare this string to another.

Parameters:

other string to compare against this one

See also:

compare(size_type, size_type, const char∗)

References compare(), and length().

5.73.3.10 int mysqlpp::String::compare (size_type *pos*, size_type *num*, const char ∗ *other*) const

Lexically compare this string to another.

Parameters:

pos position within this string to begin comparison
num maximum number of characters within this string to use in comparison
other string to compare against this one

Return values:

< 0 if this string is lexically "less than" other
0 if this string is equal to other
> 0 if this string is lexically "greater than" other

References data(), and length().

5.73.3.11 String::size_type mysqlpp::String::length () const

Return number of bytes in the string.

Note that this doesn't count the number of **characters** in the string. If your database is configured to use an 8-bit character set, this is a distinction without a difference. But, if you're using UTF-8 in the database, you will need to "widen" the UTF-8 data to use a fixed-size character set like UCS-2 and count the characters that way. You might use std::wstring, for example.

Referenced by compare().

5.73.3.12 size_type mysqlpp::String::max_size () const `[inline]`

Return the maximum number of characters in the string.

Because this is a `const` string, this is just an alias for **size()** (p. 221); its size is always equal to the amount of data currently stored.

5.73.3.13 size_type mysqlpp::String::size () const `[inline]`

Return number of bytes in string.

See commentary for **length()** (p. 220) about the difference between bytes and characters.

Referenced by at().

5.73.3.14 void mysqlpp::String::to_string (std::string & *s*) const

Copies this object's data into a C++ string.

If you know the data doesn't contain null characters (i.e. it's a typical string, not BLOB data), it's more efficient to just assign this object to anything taking `const char*`. (Or equivalently, call the **`data()`** `(p. 213)` method.) This copies a pointer to a buffer instead of copying the buffer's contents.

5.73.3.15 String& mysqlpp::String::operator= (const char * *str*) `[inline]`

Assignment operator, from C string.

This creates a copy of the entire string, not just a copy of the pointer.

References mysqlpp::mysql_type_info::string_type.

5.73.3.16 String& mysqlpp::String::operator= (const String & *other*) `[inline]`

Assignment operator, from other **String** (p. 211).

This only copies the pointer to the other String's data buffer and increments its reference counter. If you need a deep copy, assign a string to this object instead.

References buffer_.

5.73.3.17 template<typename T> bool mysqlpp::String::operator== (const T & *rhs*) const `[inline]`

Equality comparison operator.

For comparing this object to any of the data types we have a **compare()** (p. 219) overload for.

5.73.3.18 bool mysqlpp::String::operator== (const mysqlpp::null_type &) const `[inline]`

Equality comparison operator.

For checking object against MySQL++'s global `null` constant

5.73.3.19 template<typename T> bool mysqlpp::String::operator!= (const T & *rhs*) const `[inline]`

Inequality comparison operator.

For comparing this object to any of the data types we have a **compare()** (p. 219) overload for.

5.73.3.20 bool mysqlpp::String::operator!= (const mysqlpp::null_type &) const `[inline]`

Inequality comparison operator.

For checking object against MySQL++'s global `null` constant

5.73.3.21]

char mysqlpp::String::operator[ ] (**size_type** *pos*) const

Return a character within the string.

Unlike **at()** (p. 218), this access method doesn't check the index for sanity.

5.73.3.22 template<class T, class B> mysqlpp::String::operator Null< T, B > () const `[inline]`

Converts the **String** (p. 211) to a nullable data type.

This is just an implicit version of conv(Null<T, B>)

The documentation for this class was generated from the following files:

- **mystring.h**
- mystring.cpp

5.74 mysqlpp::TCPConnection Class Reference

Specialization of **`Connection`** `(p. 32)` for TCP/IP.

`#include <tcp_connection.h>`

Inheritance diagram for mysqlpp::TCPConnection:

Collaboration diagram for mysqlpp::TCPConnection:

Public Member Functions

- **TCPConnection** ()

  Create object without connecting it to the MySQL server.

- **TCPConnection** (const char ∗addr, const char ∗db=0, const char ∗user=0, const char ∗password=0)

  Create object and connect to database server over TCP/IP in one step.

- **TCPConnection** (const **TCPConnection** &other)

  Establish a new connection using the same parameters as an existing connection.

- ∼**TCPConnection** ()

  Destroy object.

- bool **connect** (const char ∗addr=0, const char ∗db=0, const char ∗user=0, const char ∗password=0)

  Connect to database after object is created.

Static Public Member Functions

- static bool **parse_address** (std::string &addr, unsigned int &port, std::string &error)

  Break the given TCP/IP address up into a separate address and port form.

5.74.1 Detailed Description

Specialization of **`Connection`** `(p. 32)` for TCP/IP.

This class just simplifies the connection creation interface of **`Connection`** `(p. 32)`. It does not add new functionality.

5.74.2 Constructor & Destructor Documentation

5.74.2.1 mysqlpp::TCPConnection::TCPConnection (const char ∗ *addr*, const char ∗ *db* = `0`, const char ∗ *user* = `0`, const char ∗ *password* = `0`) `[inline]`

Create object and connect to database server over TCP/IP in one step.

Parameters:

addr TCP/IP address of server, in either dotted quad form or as a host or domain name; may be followed by a colon and a port number or service name to override default port

db name of database to use

user user name to log in under, or 0 to use the user name the program is running under

password password to use when logging in

BEWARE: These parameters are not in the same order as those in the corresponding constructor for **Connection** (p. 32). This is a feature, not a bug. :)

5.74.2.2 mysqlpp::TCPConnection::TCPConnection (const TCPConnection & *other*) `[inline]`

Establish a new connection using the same parameters as an existing connection.

Parameters:

other pre-existing connection to clone

5.74.3 Member Function Documentation

5.74.3.1 bool mysqlpp::TCPConnection::connect (const char ∗ *addr* = `0`, const char ∗ *db* = `0`, const char ∗ *user* = `0`, const char ∗ *password* = `0`)

Connect to database after object is created.

It's better to use the connect-on-create constructor if you can. See its documentation for the meaning of these parameters.

If you call this method on an object that is already connected to a database server, the previous connection is dropped and a new connection is established.

References mysqlpp::Connection::connect(), mysqlpp::Connection::error_message_-, parse_address(), and mysqlpp::OptionalExceptions::throw_exceptions().

5.74.3.2 bool mysqlpp::TCPConnection::parse_address (std::string & *addr*, unsigned int & *port*, std::string & *error*) `[static]`

Break the given TCP/IP address up into a separate address and port form.

Does some sanity checking on the address. Only intended to try and prevent library misuse, not ensure that the address can actually be used to contact a server.

It understands the following forms:

- 1.2.3.4
- a.b.com:89
- d.e.fr:mysvcname

It also understands IPv6 addresses, but to avoid confusion between the colons they use and the colon separating the address part from the service/port part, they must be in RFC 2732 form. Example: [2010:836B:4179::836B:4179]:1234

Parameters:

addr the address and optional port/service combo to check on input, and the verified address on successful return

port the port number (resolved from the service name if necessary) on successful return

error on false return, reason for failure is placed here

Returns:

false if address fails to pass sanity checks

Referenced by connect(), and mysqlpp::Connection::parse_ipc_method().

The documentation for this class was generated from the following files:

- **tcp_connection.h**
- tcp_connection.cpp

5.75 mysqlpp::Time Class Reference

C++ form of SQL's TIME type.

`#include <datetime.h>`

Inheritance diagram for mysqlpp::Time:

Collaboration diagram for mysqlpp::Time:

Public Member Functions

- **Time** ()

  Default constructor.

- **Time** (unsigned char h, unsigned char m, unsigned char s)

  Initialize object.

- **Time** (const **Time** &other)

  *Initialize object as a copy of another **Time** (p. 228).*

- **Time** (const **DateTime** &other)

  Initialize object from time part of date/time object.

- **Time** (const char ∗str)

  Initialize object from a C string containing a SQL time string.

- template<class Str>
  Time (const Str &str)

  Initialize object from a C++ string containing a SQL time string.

- **Time** (time_t t)

  Initialize object from a `time_t`*.*

- int **compare** (const **Time** &other) const

  Compare this time to another.

- const char ∗ **convert** (const char ∗)

  Parse a SQL time string into this object.

- unsigned char **hour** () const

  Get the time's hour part, 0-23.

- void **hour** (unsigned char h)

  Change the time's hour part, 0-23.

- unsigned char **minute** () const

  Get the time's minute part, 0-59.

- void **minute** (unsigned char m)

  Change the time's minute part, 0-59.

- **operator std::string** () const

  Convert to std::string.

- **operator time_t** () const

  Convert to time_t.

- unsigned char **second** () const

  Get the time's second part, 0-59.

- void **second** (unsigned char s)

  Change the time's second part, 0-59.

- std::string **str** () const

  Return our value in std::string form.

5.75.1 Detailed Description

C++ form of SQL's TIME type.

Objects of this class can be inserted into streams, and initialized from SQL TIME strings.

5.75.2 Constructor & Destructor Documentation

5.75.2.1 mysqlpp::Time::Time (const char ∗ *str*) `[inline, explicit]`

Initialize object from a C string containing a SQL time string.

String (p. 211) must be in the HH:MM:SS format. It doesn't have to be zero-padded.

5.75.2.2 template<class Str> mysqlpp::Time::Time (const Str & *str*) `[inline, explicit]`

Initialize object from a C++ string containing a SQL time string.

This works with any stringish class that declares a c_str() member function: std::string, **mysqlpp::String** (p. 211)...

See also:

Time(const char∗) (p. 230)

5.75.2.3 mysqlpp::Time::Time (time_t *t*) `[explicit]`

Initialize object from a `time_t`.

Naturally, we throw away the "date" part of the `time_t`. If you need to keep it, you want to use **DateTime** (p. 57) instead.

5.75.3 Member Function Documentation

5.75.3.1 int mysqlpp::Time::compare (const Time & *other*) const

Compare this time to another.

Returns < 0 if this time is before the other, 0 of they are equal, and > 0 if this time is after the other.

References hour_, minute_, and second_.

5.75.3.2 mysqlpp::Time::operator time_t () const

Convert to time_t.

The "date" part of the `time_t` is "today"

The documentation for this class was generated from the following files:

- **datetime.h**
- datetime.cpp

5.76 mysqlpp::tiny_int< VT > Class Template Reference

Class for holding an SQL `TINYINT` value.

`#include <tiny_int.h>`

Collaboration diagram for mysqlpp::tiny_int< VT >:

Public Types

- typedef **tiny_int**< VT > **this_type**

  alias for this object's type

- typedef VT **value_type**

  alias for type of internal value

Public Member Functions

- **tiny_int** ()

  Default constructor.

- **tiny_int** (**value_type** v)

  Create object from any integral type that can be converted to a `short int`*.*

- **operator bool** () const

  Return truthiness of value.

- **operator int** () const

  Return value as an `int`*.*

- **operator value_type** () const

  Return raw data value with no size change.

- **this_type** & **operator=** (int v)

  Assign a new value to the object.

- **this_type** & **operator+=** (int v)

  Add another value to this object.

- **this_type** & **operator-=** (int v)

  Subtract another value to this object.

- **this_type** & **operator*=** (int v)

  Multiply this value by another object.

- **this_type** & **operator/=** (int v)

  Divide this value by another object.

- **this_type** & **operator%=** (int v)

  Divide this value by another object and store the remainder.

- **this_type** & **operator &=** (int v)

  Bitwise AND this value by another value.

- **this_type** & **operator|=** (int v)

  Bitwise OR this value by another value.

- **this_type** & **operator^=** (int v)

  Bitwise XOR this value by another value.

- **this_type** & **operator<<=** (int v)

  Shift this value left by v *positions.*

- **this_type** & **operator>>=** (int v)

  Shift this value right by v *positions.*

- **this_type** & **operator++** ()

  Add one to this value and return that value.

- **this_type** & **operator--** ()

  Subtract one from this value and return that value.

- **this_type operator++** (int)

  Add one to this value and return the previous value.

- **this_type operator--** (int)

  Subtract one from this value and return the previous value.

- **this_type operator-** (const **this_type** &i) const

  Return this value minus `i`.

- **this_type operator+** (const **this_type** &i) const

  Return this value plus `i`.

- **this_type operator*** (const **this_type** &i) const

  Return this value multiplied by `i`.

- **this_type operator/** (const **this_type** &i) const

  Return this value divided by `i`.

- **this_type operator%** (const **this_type** &i) const

  Return the modulus of this value divided by `i`.

- **this_type operator|** (const **this_type** &i) const

  Return this value bitwise OR'd by `i`.

- **this_type operator &** (const **this_type** &i) const

  Return this value bitwise AND'd by `i`.

- **this_type operator^** (const **this_type** &i) const

  Return this value bitwise XOR'd by `i`.

- **this_type operator<<** (const **this_type** &i) const

  Return this value bitwise shifted left by `i`.

- **this_type operator>>** (const **this_type** &i) const

  Return this value bitwise shifted right by `i`.

5.76.1 Detailed Description

template<typename VT = signed char> class mysqlpp::tiny_int< VT >

Class for holding an SQL `TINYINT` value.

This is required because the closest C++ type, `char`, doesn't have all the right semantics. For one, inserting a `char` into a stream won't give you a number. For another, if you don't specify signedness explicitly, C++ doesn't give a default, so it's signed on some platforms, unsigned on others.

The template parameter is intended to allow instantiating it as tiny_int<unsigned char> to hold `TINYINT UNSIGNED` values. There's nothing stopping you from using any other integer type if you want to be perverse, but please don't do that.

Several of the functions below accept an `int` argument, but internally we store the data as a `char` by default. Beware of integer overflows!

5.76.2 Constructor & Destructor Documentation

5.76.2.1 template<typename VT = signed char> mysqlpp::tiny_int< VT >::tiny_int () `[inline]`

Default constructor.

Value is uninitialized

The documentation for this class was generated from the following file:

- **tiny_int.h**

5.77 mysqlpp::TooOld< ConnInfoT > Class Template Reference

Functor to test whether a given ConnectionInfo object is "too old".

5.77.1 Detailed Description

template<typename ConnInfoT> class mysqlpp::TooOld< ConnInfoT >

Functor to test whether a given ConnectionInfo object is "too old".

The documentation for this class was generated from the following file:

- cpool.cpp

5.78 mysqlpp::Transaction Class Reference

Helper object for creating exception-safe SQL transactions.

`#include <transaction.h>`

Collaboration diagram for mysqlpp::Transaction:

Public Member Functions

- **Transaction** (**Connection** &conn, bool consistent=false)

  Constructor.

- ~**Transaction** ()

  Destructor.

- void **commit** ()

  Commits the transaction.

- void **rollback** ()

  Rolls back the transaction.

5.78.1 Detailed Description

Helper object for creating exception-safe SQL transactions.

5.78.2 Constructor & Destructor Documentation

5.78.2.1 Transaction::Transaction (Connection & *conn*, bool *consistent* = `false`)

Constructor.

Parameters:

conn The connection we use to manage the transaction set

consistent Whether to use "consistent snapshots" during the transaction. See the documentation for "START TRANSACTION" in the MySQL manual for more on this.

References mysqlpp::Connection::query().

5.78.2.2 Transaction::∼Transaction ()

Destructor.

If the transaction has not been committed or rolled back by the time the destructor is called, it is rolled back. This is the right thing because one way this can happen is if the object is being destroyed as the stack is unwound to handle an exception. In that instance, you certainly want to roll back the transaction.

References rollback().

5.78.3 Member Function Documentation

5.78.3.1 void Transaction::commit ()

Commits the transaction.

This commits all updates to the database using the connection we were created with since this object was created. This is a no-op if the table isn't stored using a transaction-aware storage engine. See CREATE TABLE in the MySQL manual for details.

References mysqlpp::Query::execute(), and mysqlpp::Connection::query().

5.78.3.2 void Transaction::rollback ()

Rolls back the transaction.

This abandons all SQL statements made on the connection since this object was created. This only works on tables stored using a transaction-aware storage engine. See CREATE TABLE in the MySQL manual for details.

References mysqlpp::Query::execute(), and mysqlpp::Connection::query().

Referenced by ∼Transaction().

The documentation for this class was generated from the following files:

- **transaction.h**
- transaction.cpp

5.79 mysqlpp::TypeLookupFailed Class Reference

Thrown from the C++ to SQL data type conversion routine when it can't figure out how to map the type.

`#include <exceptions.h>`

Inheritance diagram for mysqlpp::TypeLookupFailed:

Collaboration diagram for mysqlpp::TypeLookupFailed:

Public Member Functions

- **TypeLookupFailed** (const std::string &w)

  Create exception object.

5.79.1 Detailed Description

Thrown from the C++ to SQL data type conversion routine when it can't figure out how to map the type.

This exception is not optional. The only alternatives when this happens are equally drastic: basically, either iterate past the end of an array (crashing the program) or call assert() to crash the program nicely. At least this way you have some control over how your program ends. You can even ignore the error and keep on going: this typically happens when building a SQL query, so you can handle it just the same as if the subsequent query execution failed.

The documentation for this class was generated from the following file:

- **exceptions.h**

5.80 mysqlpp::UnixDomainSocketConnection Class Reference

Specialization of **`Connection`** `(p. 32)` for Unix domain sockets.

`#include <uds_connection.h>`

Inheritance diagram for mysqlpp::UnixDomainSocketConnection:

Collaboration diagram for mysqlpp::UnixDomainSocketConnection:

Public Member Functions

- **UnixDomainSocketConnection** ()

  Create object without connecting it to the MySQL server.

- **UnixDomainSocketConnection** (const char ∗path, const char ∗db=0, const char ∗user=0, const char ∗password=0)

  Create object and connect to database server over Unix domain sockets in one step.

- **UnixDomainSocketConnection** (const **UnixDomainSocketConnection** &other)

  Establish a new connection using the same parameters as an existing connection.

- ∼**UnixDomainSocketConnection** ()

Destroy object.

- bool **connect** (const char ∗path, const char ∗db=0, const char ∗user=0, const char ∗password=0)

  Connect to database after object is created.

Static Public Member Functions

- static bool **is_socket** (const char ∗path, std::string ∗error=0)

  Check that the given path names a Unix domain socket and that we have read-write permission for it.

5.80.1 Detailed Description

Specialization of **`Connection`** `(p. 32)` for Unix domain sockets.

This class just simplifies the connection creation interface of **`Connection`** `(p. 32)`. It does not add new functionality.

5.80.2 Constructor & Destructor Documentation

5.80.2.1 mysqlpp::UnixDomainSocketConnection::UnixDomainSocketConnection (const char ∗ *path*, const char ∗ *db* = `0`, const char ∗ *user* = `0`, const char ∗ *password* = `0`) `[inline]`

Create object and connect to database server over Unix domain sockets in one step.

Parameters:

path filesystem path to socket

db name of database to use

user user name to log in under, or 0 to use the user name the program is running under

password password to use when logging in

BEWARE: These parameters are not in the same order as those in the corresponding constructor for **Connection** (p. 32). This is a feature, not a bug. :)

References connect().

5.80.2.2 mysqlpp::UnixDomainSocketConnection::UnixDomainSocketConnection (const UnixDomainSocketConnection & *other*) `[inline]`

Establish a new connection using the same parameters as an existing connection.

Parameters:

other pre-existing connection to clone

5.80.3 Member Function Documentation

5.80.3.1 bool mysqlpp::UnixDomainSocketConnection::connect (const char ∗ *path*, const char ∗ *db* = `0`, const char ∗ *user* = `0`, const char ∗ *password* = `0`)

Connect to database after object is created.

It's better to use the connect-on-create constructor if you can. See its documentation for the meaning of these parameters.

If you call this method on an object that is already connected to a database server, the previous connection is dropped and a new connection is established.

References mysqlpp::Connection::connect(), mysqlpp::Connection::error_message_-
, is_socket(), and mysqlpp::OptionalExceptions::throw_exceptions().

Referenced by UnixDomainSocketConnection().

5.80.3.2 bool mysqlpp::UnixDomainSocketConnection::is_socket (const char ∗ *path*, std::string ∗ *error* = `0`) `[static]`

Check that the given path names a Unix domain socket and that we have read-write permission for it.

Parameters:

path the filesystem path to the socket

error on failure, reason is placed here; take default if you do not need a reason if it fails

Returns:

false if address fails to pass sanity checks

Referenced by connect(), and mysqlpp::Connection::parse_ipc_method().

The documentation for this class was generated from the following files:

- **uds_connection.h**
- uds_connection.cpp

5.81 mysqlpp::UseEmbeddedConnectionOption Class Reference

Connect to embedded server in preference to remote server.

`#include <options.h>`

Inheritance diagram for mysqlpp::UseEmbeddedConnectionOption:

Collaboration diagram for mysqlpp::UseEmbeddedConnectionOption:

5.81.1 Detailed Description

Connect to embedded server in preference to remote server.

The documentation for this class was generated from the following file:

- **options.h**

5.82 mysqlpp::UseQueryError Class Reference

Exception (p. 84) thrown when something goes wrong in processing a "use" query.

`#include <exceptions.h>`

Inheritance diagram for mysqlpp::UseQueryError:

Collaboration diagram for mysqlpp::UseQueryError:

Public Member Functions

- **UseQueryError** (const char ∗w="")

  Create exception object.

5.82.1 Detailed Description

Exception (p. 84) thrown when something goes wrong in processing a "use" query.

The documentation for this class was generated from the following file:

- **exceptions.h**

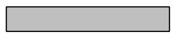
5.83 mysqlpp::UseQueryResult Class Reference

StoreQueryResult (p. 209) set type for "use" queries.

```
#include <result.h>
```

Inheritance diagram for mysqlpp::UseQueryResult:

Collaboration diagram for mysqlpp::UseQueryResult:



Public Member Functions

- **UseQueryResult** ()

  Default constructor.

- **UseQueryResult** (MYSQL_RES ∗result, **DBDriver** ∗dbd, bool te=true)

  Create the object, fully initialized.

- **UseQueryResult** (const **UseQueryResult** &other)

  Create a copy of another ***UseQueryResult*** *(p. 247) object.*

- ∼**UseQueryResult** ()

  Destroy object.

- **UseQueryResult** & **operator=** (const **UseQueryResult** &rhs)

  Copy another ***UseQueryResult*** *(p. 247) object's data into this object.*

- const **Field** & **fetch_field** () const

Returns the next field in this result set.

- const **Field** & **fetch_field** (Fields::size_type i) const
  Returns the given field in this result set.
- const unsigned long ∗ **fetch_lengths** () const
  Returns the lengths of the fields in the current row of the result set.
- **Row fetch_row** () const
  Returns the next row in a "use" query's result set.
- MYSQL_ROW **fetch_raw_row** () const
  Wraps mysql_fetch_row() in MySQL C API.
- void **field_seek** (Fields::size_type field) const
  Jumps to the given field within the result set.
- **operator MYSQL_RES** ∗ () const
  Return the pointer to the underlying MySQL C API result set object.

5.83.1 Detailed Description

StoreQueryResult (p. 209) set type for "use" queries.

See the user manual for the reason you might want to use this even though its interface is less friendly than StoreQueryResult's.

5.83.2 Member Function Documentation

5.83.2.1 Row mysqlpp::UseQueryResult::fetch_row () const

Returns the next row in a "use" query's result set.

This is a thick wrapper around **DBDriver::fetch_row()** (p. 70). It does a lot of error checking before returning the **Row** (p. 166) object containing the row data.

See also:

fetch_raw_row() (p. 248)

References mysqlpp::ResultBase::driver_, fetch_lengths(), mysqlpp::DBDriver::fetch_row(), mysqlpp::RefCountedPointer< T, Destroyer >::raw(), and mysqlpp::OptionalExceptions::throw_exceptions().

Referenced by mysqlpp::Query::for_each(), and mysqlpp::Query::store_if().

5.83.2.2 void mysqlpp::UseQueryResult::field_seek (Fields::size_type *field*) const [inline]

Jumps to the given field within the result set.

Calling this allows you to reset the default field index used by **fetch_field()** (p. 163).

5.83.2.3 mysqlpp::UseQueryResult::operator MYSQL_RES ∗ () const [inline]

Return the pointer to the underlying MySQL C API result set object.

While this has obvious inherent value for those times you need to dig beneath the MySQL++ interface, it has subtler value. It effectively stands in for operator bool(), operator !(), operator ==(), and operator !=(), because the C++ compiler can implement all of these with a MYSQL_RES∗.

Of these uses, the most valuable is using the **UseQueryResult** (p. 247) object in bool context to determine if the query that created

```
Query q("....");
if (UseQueryResult res = q.use()) {
    // Can use 'res', query succeeded
}
else {
    // Query failed, call Query::error() or ::errnum() for why
}
```

The documentation for this class was generated from the following files:

- **result.h**
- result.cpp

5.84 mysqlpp::UseRemoteConnectionOption Class Reference

Connect to remote server in preference to embedded server.

`#include <options.h>`

Inheritance diagram for mysqlpp::UseRemoteConnectionOption:

Collaboration diagram for mysqlpp::UseRemoteConnectionOption:

5.84.1 Detailed Description

Connect to remote server in preference to embedded server.

The documentation for this class was generated from the following file:

- **options.h**

5.85 mysqlpp::value_list_b< Seq, Manip > Struct Template Reference

Same as **value_list_ba** (p. 253), plus the option to have some elements of the list suppressed.

`#include <vallist.h>`

Collaboration diagram for mysqlpp::value_list_b< Seq, Manip >:

Public Member Functions

- **value_list_b** (const Seq &s, const std::vector< bool > &f, const char ∗d, Manip m)

  Create object.

Public Attributes

- const Seq ∗ **list**

  set of objects in the value list

- const std::vector< bool > **fields**

  delimiter to use between each value in the list when inserting it into a C++ stream

- const char ∗ **delim**

  delimiter to use between each value in the list when inserting it into a C++ stream

- Manip **manip**

  manipulator to use when inserting the list into a C++ stream

5.85.1 Detailed Description

template<class Seq, class Manip> struct mysqlpp::value_list_b< Seq, Manip >

Same as **value_list_ba** (p. 253), plus the option to have some elements of the list suppressed.

Imagine an object of this type contains the list (a, b, c), that the object's 'fields' list is (true, false, true), and that the object's delimiter is set to ":". When you insert that object into a C++ stream, you would get "a:c".

See value_list_ba's documentation for more details.

5.85.2 Constructor & Destructor Documentation

5.85.2.1 template<class Seq, class Manip> mysqlpp::value_list_b< Seq, Manip >::value_list_b (const Seq & *s*, const std::vector< bool > & *f*, const char * *d*, Manip *m*) `[inline]`

Create object.

Parameters:

s set of objects in the value list

f for each true item in the list, the list item in that position will be inserted into a C++ stream

d what delimiter to use between each value in the list when inserting the list into a C++ stream

m manipulator to use when inserting the list into a C++ stream

The documentation for this struct was generated from the following file:

- **vallist.h**

5.86 mysqlpp::value_list_ba< Seq, Manip > Struct Template Reference

Holds a list of items, typically used to construct a SQL "value list".

`#include <vallist.h>`

Collaboration diagram for mysqlpp::value_list_ba< Seq, Manip >:

Public Member Functions

- **value_list_ba** (const Seq &s, const char ∗d, Manip m)

  Create object.

Public Attributes

- const Seq ∗ **list**

  set of objects in the value list

- const char ∗ **delim**

  delimiter to use between each value in the list when inserting it into a C++ stream

- Manip **manip**

  manipulator to use when inserting the list into a C++ stream

5.86.1 Detailed Description

template<class Seq, class Manip> struct mysqlpp::value_list_ba< Seq, Manip >

Holds a list of items, typically used to construct a SQL "value list".

The SQL INSERT statement has a VALUES clause; this class can be used to construct the list of items for that clause.

Imagine an object of this type contains the list (a, b, c), and that the object's delimiter symbol is set to ", ". When you insert that object into a C++ stream, you would get "a, b, c".

This class is never instantiated by hand. The value_list() functions build instances of this structure template to do their work. MySQL++'s SSQLS mechanism calls those functions when building SQL queries; you can call them yourself to do similar work. The "Harnessing SSQLS Internals" section of the user manual has some examples of this.

See also:

value_list_b (p. 251)

5.86.2 Constructor & Destructor Documentation

5.86.2.1 template<class Seq, class Manip> mysqlpp::value_list_ba< Seq, Manip >::value_list_ba (const Seq & *s*, const char ∗ *d*, Manip *m*) `[inline]`

Create object.

Parameters:

s set of objects in the value list

d what delimiter to use between each value in the list when inserting the list into a C++ stream

m manipulator to use when inserting the list into a C++ stream

The documentation for this struct was generated from the following file:

- **vallist.h**

5.87 mysqlpp::WindowsNamedPipeConnection Class Reference

Specialization of **`Connection`** `(p. 32)` for Windows named pipes.

`#include <wnp_connection.h>`

Inheritance diagram for mysqlpp::WindowsNamedPipeConnection:

Collaboration diagram for mysqlpp::WindowsNamedPipeConnection:

Public Member Functions

- **WindowsNamedPipeConnection** ()

  Create object without connecting it to the MySQL server.

- **WindowsNamedPipeConnection** (const char ∗db, const char ∗user=0, const char ∗password=0)

  Create object and connect to database server over Windows named pipes in one step.

- **WindowsNamedPipeConnection** (const **WindowsNamedPipeConnection** &other)

  Establish a new connection using the same parameters as an existing connection.

- ∼**WindowsNamedPipeConnection** ()

Destroy object.

- bool **connect** (const char ∗db=0, const char ∗user=0, const char ∗password=0)

  Connect to database after object is created.

Static Public Member Functions

- static bool **is_wnp** (const char ∗server)

  Check that given string denotes a Windows named pipe connection to MySQL.

5.87.1 Detailed Description

Specialization of **`Connection`** `(p. 32)` for Windows named pipes.

This class just simplifies the connection creation interface of **`Connection`** `(p. 32)`. It does not add new functionality.

5.87.2 Constructor & Destructor Documentation

5.87.2.1 mysqlpp::WindowsNamedPipeConnection::WindowsNamedPipeConnection (const char ∗ *db*, const char ∗ *user* = `0`, const char ∗ *password* = `0`) `[inline]`

Create object and connect to database server over Windows named pipes in one step.

Parameters:

db name of database to use

user user name to log in under, or 0 to use the user name the program is running under

password password to use when logging in

5.87.2.2 mysqlpp::WindowsNamedPipeConnection::WindowsNamedPipeConnection (const WindowsNamedPipeConnection & *other*) `[inline]`

Establish a new connection using the same parameters as an existing connection.

Parameters:

other pre-existing connection to clone

5.87.3 Member Function Documentation

5.87.3.1 bool mysqlpp::WindowsNamedPipeConnection::connect (const char ∗ *db* = `0`, const char ∗ *user* = `0`, const char ∗ *password* = `0`)

Connect to database after object is created.

It's better to use the connect-on-create constructor if you can. See its documentation for the meaning of these parameters.

If you call this method on an object that is already connected to a database server, the previous connection is dropped and a new connection is established.

References mysqlpp::Connection::connect(), mysqlpp::Connection::error_message_-, and mysqlpp::OptionalExceptions::throw_exceptions().

5.87.3.2 bool mysqlpp::WindowsNamedPipeConnection::is_wnp (const char ∗ *server*) `[static]`

Check that given string denotes a Windows named pipe connection to MySQL.

Parameters:

server the server address

Returns:

false if server address does not denote a Windows named pipe connection, or we are not running on Windows

Referenced by mysqlpp::Connection::parse_ipc_method().

The documentation for this class was generated from the following files:

- **wnp_connection.h**
- wnp_connection.cpp

5.88 mysqlpp::WriteTimeoutOption Class Reference

Set (p. 182) timeout for IPC data reads.

`#include <options.h>`

Inheritance diagram for mysqlpp::WriteTimeoutOption:

Collaboration diagram for mysqlpp::WriteTimeoutOption:

5.88.1 Detailed Description

Set (p. 182) timeout for IPC data reads.

The documentation for this class was generated from the following file:

- **options.h**

Chapter 6

File Documentation

6.1 autoflag.h File Reference

Defines a template for setting a flag within a given variable scope, and resetting it when exiting that scope.

Classes

- class **AutoFlag**< **T** >

  A template for setting a flag on a variable as long as the object that set it is in scope. Flag resets when object goes out of scope. Works on anything that looks like bool.

6.1.1 Detailed Description

Defines a template for setting a flag within a given variable scope, and resetting it when exiting that scope.

6.2 beemutex.h File Reference

MUTually EXclusive lock class.

`#include "exceptions.h"`

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::BeecryptMutex**
  Wrapper around platform-specific mutexes.
- class **mysqlpp::ScopedLock**
  *Wrapper around **BeecryptMutex** (p. 26) to add scope-bound locking and unlocking.*

6.2.1 Detailed Description

MUTually EXclusive lock class.

Author:

Bob Deblier `<bob.deblier@telenet.be>`

Modified by Warren Young of Educational Technology Resources, Inc. from version in Beecrypt 4.1.2:

- minor style changes to make it fit within MySQL++
- changed init() to a ctor and destroy() to a dtor
- class just becomes a no-op if no supported mutex type is available
- throwing MutexFailed instead of `char*`
- moved all method implementations from inline in the .h file to a .cpp file so we don't have to make the header depend on config.h on autoconf-using systems
- made private mutex member a void* so we don't have to define the full type in the .h file, due to previous item
- added more Doxygen comments, and changed some existing comments

6.3 common.h File Reference

This file includes top-level definitions for use both internal to the library, and outside it. Contrast mysql++.h.

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

6.3.1 Detailed Description

This file includes top-level definitions for use both internal to the library, and outside it. Contrast mysql++.h.

This file mostly takes care of platform differences.

6.4 comparable.h File Reference

Declares the Comparable<T> mixin.

Classes

- class **Comparable**< **T** >

  Mix-in that gives its subclass a full set of comparison operators.

6.4.1 Detailed Description

Declares the Comparable<T> mixin.

6.5 connection.h File Reference

Declares the Connection class.

```
#include "common.h"
#include "noexceptions.h"
#include "options.h"
#include <string>
#include "tcp_connection.h"
#include "uds_connection.h"
#include "wnp_connection.h"
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::Connection**
  Manages the connection to the database server.

6.5.1 Detailed Description

Declares the Connection class.

Every program using MySQL++ must create a Connection object, which manages information about the connection to the database server, and performs connection-related operations once the connection is up. Subordinate classes, such as Query and Row take their defaults as to whether exceptions are thrown when errors are encountered from the Connection object that created them, directly or indirectly.

6.6 cpool.h File Reference

Declares the ConnectionPool class.

```
#include "beemutex.h"
#include <list>
#include <assert.h>
#include <time.h>
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::ConnectionPool**

  Manages a pool of connections for programs that need more than one ***Connection*** *(p. 32) object at a time, but can't predict how many they need in advance.*

- struct **mysqlpp::ConnectionPool::ConnectionInfo**

6.6.1 Detailed Description

Declares the ConnectionPool class.

6.7 custom.h File Reference

Backwards-compatibility header; loads ssqls.h.

```
#include "ssqls.h"
```

6.7.1 Detailed Description

Backwards-compatibility header; loads ssqls.h.

6.8 datetime.h File Reference

Declares classes to add SQL-compatible date and time types to C++'s type system.

```
#include "common.h"
#include "comparable.h"
#include <string>
#include <iostream>
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::DateTime**
  C++ form of SQL's DATETIME type.
- class **mysqlpp::Date**
  C++ form of SQL's DATE type.
- class **mysqlpp::Time**
  C++ form of SQL's TIME type.

Functions

- std::ostream & **mysqlpp::operator**<< (std::ostream &os, const DateTime &dt)
  *Inserts a **DateTime** (p. 57) object into a C++ stream in a SQL-compatible format.*
- std::ostream & **mysqlpp::operator**<< (std::ostream &os, const Date &d)
  *Inserts a **Date** (p. 53) object into a C++ stream.*
- std::ostream & **mysqlpp::operator**<< (std::ostream &os, const Time &t)
  *Inserts a **Time** (p. 228) object into a C++ stream in a SQL-compatible format.*

6.8.1 Detailed Description

Declares classes to add SQL-compatible date and time types to C++'s type system.

6.9 dbdriver.h File Reference

Declares the DBDriver class.

```
#include "common.h"
#include "options.h"
#include <typeinfo>
#include <limits.h>
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::DBDriver**
  Provides a thin abstraction layer over the underlying database client library.

6.9.1 Detailed Description

Declares the DBDriver class.

6.10 exceptions.h File Reference

Declares the MySQL++-specific exception classes.

```
#include "options.h"
#include <exception>
#include <string>
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::Exception**

  Base class for all MySQL++ custom exceptions.

- class **mysqlpp::BadConversion**

  ***Exception** (p. 84) thrown when a bad type conversion is attempted.*

- class **mysqlpp::BadFieldName**

  ***Exception** (p. 84) thrown when a requested named field doesn't exist.*

- class **mysqlpp::BadOption**

  ***Exception** (p. 84) thrown when you pass an unrecognized option to **Connection::set_-option()** (p. 40).*

- class **mysqlpp::BadParamCount**

  ***Exception** (p. 84) thrown when not enough query parameters are provided.*

- class **mysqlpp::UseQueryError**

  ***Exception** (p. 84) thrown when something goes wrong in processing a "use" query.*

- class **mysqlpp::BadQuery**

  ***Exception** (p. 84) thrown when the database server encounters a problem while processing your query.*

- class **mysqlpp::ConnectionFailed**

  ***Exception** (p. 84) thrown when there is a problem related to the database server connection.*

- class **mysqlpp::DBSelectionFailed**

__Exception__ (p. 84) thrown when the program tries to select a new database and the database server refuses for some reason.

- class **mysqlpp::MutexFailed**

  __Exception__ (p. 84) thrown when a __BeecryptMutex__ (p. 26) object fails.

- class **mysqlpp::ObjectNotInitialized**

  __Exception__ (p. 84) thrown when you try to use an object that isn't completely initialized.

- class **mysqlpp::SelfTestFailed**

  Used within MySQL++'s test harness only.

- class **mysqlpp::TypeLookupFailed**

  Thrown from the C++ to SQL data type conversion routine when it can't figure out how to map the type.

6.10.1 Detailed Description

Declares the MySQL++-specific exception classes.

When exceptions are enabled for a given **mysqlpp::OptionalExceptions** (p. 123) derivative, any of these exceptions can be thrown on error.

6.11 field.h File Reference

Declares the Field and Fields classes.

```
#include "common.h"
#include "type_info.h"
#include <vector>
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::Field**
  Class to hold information about a SQL field.

Typedefs

- typedef std::vector< Field > **mysqlpp::Fields**
  The list-of-Fields type.

6.11.1 Detailed Description

Declares the Field and Fields classes.

6.12 field_names.h File Reference

Declares a class to hold a list of field names.

```
#include <string>
#include <vector>
#include <ctype.h>
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::FieldNames**
  Holds a list of SQL field names.

6.12.1 Detailed Description

Declares a class to hold a list of field names.

6.13 field_types.h File Reference

Declares a class to hold a list of SQL field type info.

```
#include "type_info.h"
```

```
#include <vector>
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::FieldTypes**

  A vector of SQL field types.

6.13.1 Detailed Description

Declares a class to hold a list of SQL field type info.

6.14 manip.h File Reference

Declares the Query stream manipulators and operators.

```
#include "common.h"
#include "myset.h"
#include "stadapter.h"
#include <iostream>
```

Namespaces

- namespace **mysqlpp**

Enumerations

- enum **quote_type0** { **mysqlpp::quote** }
- enum **quote_only_type0** { **mysqlpp::quote_only** }
- enum **quote_double_only_type0** { **mysqlpp::quote_double_only** }
- enum **escape_type0** { **escape** }
- enum **do_nothing_type0** { **mysqlpp::do_nothing** }
- enum **ignore_type0** { **mysqlpp::ignore** }

Functions

- SQLQueryParms & **mysqlpp::operator**<< (escape_type2 p, SQLTypeAdapter &in)

  Inserts a ***SQLTypeAdapter*** (p. *198) into a stream, escaping special SQL characters.*

- ostream & **mysqlpp::operator**<< (escape_type1 o, const SQLTypeAdapter &in)

  Inserts anything that can be converted to ***SQLTypeAdapter*** (p. *198) into a stream, escaping special SQL characters as needed.*

6.14.1 Detailed Description

Declares the Query stream manipulators and operators.

These manipulators let you automatically quote elements or escape characters that are special in SQL when inserting them into a Query stream. They make it easier to build syntactically-correct SQL queries.

This file also includes special `operator<<` definitions for a few key MySQL++ data types, since we know when to do automatic quoting and escaping for these types. This only works with Query streams, not regular std::ostreams, since we're only concerned with making correct SQL, not with presentation matters.

test/test_manip.cpp exercises the mechanisms defined here.

6.15 myset.h File Reference

Declares templates for generating custom containers used elsewhere in the library.

```
#include "common.h"
#include "mystring.h"
#include "stream2string.h"
#include <iostream>
#include <set>
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::Set**< **Container** >

  A special std::set derivative for holding MySQL data sets.

Functions

- template<class Container>
  std::ostream & **mysqlpp::operator**<< (std::ostream &s, const Set< Container > &d)

  Inserts a ***Set*** *(*p. *182) object into a C++ stream.*

6.15.1 Detailed Description

Declares templates for generating custom containers used elsewhere in the library.

6.16 mysql++.h File Reference

The main MySQL++ header file.

```
#include "connection.h"
#include "cpool.h"
#include "query.h"
#include "sql_types.h"
#include "transaction.h"
```

Namespaces

- namespace **mysqlpp**

Defines

- #define **MYSQLPP_VERSION**(major, minor, bugfix) (((major) << 16) | ((minor) << 8) | (bugfix))

  Encode MySQL++ library version number.

- #define **MYSQLPP_HEADER_VERSION** MYSQLPP_VERSION(3, 0, 6)

  Get the library version number that mysql++.h comes from.

Functions

- unsigned int **mysqlpp::get_library_version** ()

  Get the current MySQL++ library version number.

6.16.1 Detailed Description

The main MySQL++ header file.

This file brings in all MySQL++ headers except for **custom.h** (p. 265) and custom-macros.h which are a strictly optional feature of MySQL++.

There is no point in trying to optimize which headers you include, because the MySQL++ headers are so intertwined. You can only get trivial compile time benefits, at the expense of clarity.

6.16.2 Define Documentation

6.16.2.1 #define MYSQLPP_HEADER_VERSION MYSQLPP_VERSION(3, 0, 6)

Get the library version number that mysql++.h comes from.

MySQL++ Version number that the mysql++.h header file comes from, encoded by MYSQLPP_VERSION macro. Compare this value to what mysqlpp_lib_version() returns in order to ensure that your program is using header files from the same version of MySQL++ as the actual library you're linking to.

6.16.2.2 #define MYSQLPP_VERSION(major, minor, bugfix) (((major) << 16) | ((minor) << 8) | (bugfix))

Encode MySQL++ library version number.

This macro takes major, minor and bugfix numbers (e.g. 1, 2, and 3) and encodes them like 0x010203.

6.17 mystring.h File Reference

Declares String class, MySQL++'s generic std::string-like class, used for holding data received from the database server.

```
#include "common.h"
#include "datetime.h"
#include "exceptions.h"
#include "null.h"
#include "sql_buffer.h"
#include <string>
#include <sstream>
#include <limits>
#include <stdlib.h>
#include <string.h>
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::String**

  A std::string work-alike that can convert itself from SQL text data formats to C++ data types.

Functions

- std::ostream & **mysqlpp::operator**<< (std::ostream &o, const String &in)

  Stream insertion operator for ***String*** *(p. 211) objects.*

6.17.1 Detailed Description

Declares String class, MySQL++'s generic std::string-like class, used for holding data received from the database server.

6.18 noexceptions.h File Reference

Declares interface that allows exceptions to be optional.

```
#include "common.h"
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::OptionalExceptions**

  Interface allowing a class to have optional exceptions.

- class **mysqlpp::NoExceptions**

  Disable exceptions in an object derived from ***OptionalExceptions*** *(p. 123).*

6.18.1 Detailed Description

Declares interface that allows exceptions to be optional.

A class may inherit from OptionalExceptions, which will add to it a mechanism by which a user can tell objects of that class to suppress exceptions. (They are enabled by default.) This module also declares a NoExceptions class, objects of which take a reference to any class derived from OptionalExceptions. The NoExceptions constructor calls the method that disables exceptions, and the destructor reverts them to the previous state. One uses the NoExceptions object within a scope to suppress exceptions in that block, without having to worry about reverting the setting when the block exits.

6.19 null.h File Reference

Declares classes that implement SQL "null" semantics within C++'s type system.

```
#include "exceptions.h"
#include <iostream>
#include <string>
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::null_type**

  The type of the global mysqlpp::null object.

- struct **mysqlpp::NullIsNull**

  *Class for objects that define SQL null in terms of MySQL++'s **null_type** (p. 116).*

- struct **mysqlpp::NullIsZero**

  Class for objects that define SQL null as 0.

- struct **mysqlpp::NullIsBlank**

  Class for objects that define SQL null as a blank C string.

- class **mysqlpp::Null**< **Type, Behavior** >

  Class for holding data from a SQL column with the NULL attribute.

Functions

- template<class Type, class Behavior>
  std::ostream & **mysqlpp::operator**<< (std::ostream &o, const Null< Type, Behavior > &n)

  Inserts null-able data into a C++ stream if it is not actually null. Otherwise, insert something appropriate for null data.

Variables

- const std::string **mysqlpp::null_str**

  "NULL" string constant

- const null_type **mysqlpp::null** = null_type()

  Global 'null' instance. Use wherever you need a SQL null.

6.19.1 Detailed Description

Declares classes that implement SQL "null" semantics within C++'s type system.

This is required because C++'s own NULL type is not semantically the same as SQL nulls.

6.20 options.h File Reference

Declares the Option class hierarchy, used to implement connection options in Connection and DBDriver classes.

```
#include "common.h"
#include <deque>
#include <string>
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::Option**

  Define abstract interface for all ∗Option subclasses.

- class **mysqlpp::DataOption**< **T** >

  Define abstract interface for all ∗Options that take a lone scalar as an argument.

- class **mysqlpp::CompressOption**

  Enable data compression on the connection.

- class **mysqlpp::ConnectTimeoutOption**

  *Change **Connection::connect()** (p. 37) default timeout.*

- class **mysqlpp::FoundRowsOption**

  *Make **Query::affected_rows()** (p. 126) return number of matched rows.*

- class **mysqlpp::GuessConnectionOption**

  Allow C API to guess what kind of connection to use.

- class **mysqlpp::IgnoreSpaceOption**

  Allow spaces after function names in queries.

- class **mysqlpp::InitCommandOption**

  Give SQL executed on connect.

- class **mysqlpp::InteractiveOption**

  Assert that this is an interactive program.

- class **mysqlpp::LocalFilesOption**

  Enable LOAD DATA LOCAL statement.

- class **mysqlpp::LocalInfileOption**

  Enable LOAD LOCAL INFILE statement.

- class **mysqlpp::MultiResultsOption**

  Enable multiple result sets in a reply.

- class **mysqlpp::MultiStatementsOption**

  Enable multiple queries in a request to the server.

- class **mysqlpp::NamedPipeOption**

  Suggest use of named pipes.

- class **mysqlpp::NoSchemaOption**

  Disable db.tbl.col syntax in queries.

- class **mysqlpp::ReadDefaultFileOption**

  Override use of my.cnf.

- class **mysqlpp::ReadDefaultGroupOption**

  Override use of my.cnf.

- class **mysqlpp::ReadTimeoutOption**

  ***Set** (p. 182) timeout for IPC data reads.*

- class **mysqlpp::ReconnectOption**

  Enable automatic reconnection to server.

- class **mysqlpp::ReportDataTruncationOption**

  ***Set** (p. 182) reporting of data truncation errors.*

- class **mysqlpp::SecureAuthOption**

  Enforce use of secure authentication, refusing connection if not available.

- class **mysqlpp::SetCharsetDirOption**

  Give path to charset definition files.

- class **mysqlpp::SetCharsetNameOption**

  Give name of default charset.

- class **mysqlpp::SetClientIpOption**

  Fake client IP address when connecting to embedded server.

- class **mysqlpp::SharedMemoryBaseNameOption**

  Set *(*p. *182) name of shmem segment for IPC.*

- class **mysqlpp::SslOption**

  Specialized option for handling SSL parameters.

- class **mysqlpp::UseEmbeddedConnectionOption**

  Connect to embedded server in preference to remote server.

- class **mysqlpp::UseRemoteConnectionOption**

  Connect to remote server in preference to embedded server.

- class **mysqlpp::WriteTimeoutOption**

  Set *(*p. *182) timeout for IPC data reads.*

Typedefs

- typedef DataOption< unsigned > **mysqlpp::IntegerOption**

  Option *(*p. *121) w/ int argument.*

- typedef DataOption< bool > **mysqlpp::BooleanOption**

  Option *(*p. *121) w/ bool argument.*

- typedef DataOption< std::string > **mysqlpp::StringOption**

  Option *(*p. *121) w/ string argument.*

- typedef std::deque< Option ∗ > **mysqlpp::OptionList**

  The data type of the list of connection options.

- typedef OptionList::const_iterator **mysqlpp::OptionListIt**

  Primary iterator type into List.

6.20.1 Detailed Description

Declares the Option class hierarchy, used to implement connection options in Connection and DBDriver classes.

This is tied closely enough to DBDriver that there's a pure-OO argument that it should be declared as protected or private members within DBDriver. We do it outside DBDriver because there's so much of it. It'd overwhelm everything else that's going on in that class totally out of proprortion to the importance of options.

6.21 qparms.h File Reference

Declares the template query parameter-related stuff.

```
#include "stadapter.h"
#include <vector>
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::SQLQueryParms**

  This class holds the parameter values for filling template queries.

- struct **mysqlpp::SQLParseElement**

  *Used within **Query** (p. 126) to hold elements for parameterized queries.*

6.21.1 Detailed Description

Declares the template query parameter-related stuff.

The classes defined in this file are used by class Query when it parses a template query: they hold information that it finds in the template, so it can assemble a SQL statement later on demand.

6.22 query.h File Reference

Defines a class for building and executing SQL queries.

```
#include "common.h"
#include "noexceptions.h"
#include "qparms.h"
#include "querydef.h"
#include "result.h"
#include "row.h"
#include "stadapter.h"
#include <deque>
#include <iomanip>
#include <list>
#include <map>
#include <set>
#include <vector>
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::Query**
  A class for building and executing SQL queries.

Functions

- std::ostream & **mysqlpp::operator**<< (std::ostream &os, Query &q)
  Insert raw query string into the given stream.

6.22.1 Detailed Description

Defines a class for building and executing SQL queries.

6.23 refcounted.h File Reference

Declares the RefCountedPointer template.

```
#include <memory>
```

Namespaces

- namespace **mysqlpp**

Classes

- struct **mysqlpp::RefCountedPointerDestroyer**< **T** >
  Functor to call delete on the pointer you pass to it.
- class **mysqlpp::RefCountedPointer**< **T, Destroyer** >
  Creates an object that acts as a reference-counted pointer to another object.

6.23.1 Detailed Description

Declares the RefCountedPointer template.

6.24 result.h File Reference

Declares classes for holding information about SQL query results.

```
#include "common.h"
#include "exceptions.h"
#include "field.h"
#include "field_names.h"
#include "field_types.h"
#include "noexceptions.h"
#include "refcounted.h"
#include "row.h"
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::SimpleResult**

  Holds information about the result of queries that don't return rows.

- class **mysqlpp::ResultBase**

  Base class for ***StoreQueryResult*** *(p. 209) and* ***UseQueryResult*** *(p. 247).*

- class **mysqlpp::StoreQueryResult**

  StoreQueryResult *(p. 209) set type for "store" queries.*

- struct **mysqlpp::RefCountedPointerDestroyer**< **MYSQL_RES** >

  Functor to call mysql_free_result() on the pointer you pass to it.

- class **mysqlpp::UseQueryResult**

  StoreQueryResult *(p. 209) set type for "use" queries.*

Functions

- void **mysqlpp::swap** (StoreQueryResult &x, StoreQueryResult &y)

  Swaps two ***StoreQueryResult*** *(p. 209) objects.*

- void **mysqlpp::swap** (UseQueryResult &x, UseQueryResult &y)

  *Swaps two **UseQueryResult** (p. 247) objects.*

6.24.1 Detailed Description

Declares classes for holding information about SQL query results.

6.25 row.h File Reference

Declares the classes for holding row data from a result set.

```
#include "common.h"
#include "mystring.h"
#include "noexceptions.h"
#include "refcounted.h"
#include "vallist.h"
#include <vector>
#include <string>
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::Row**

    Manages rows from a result set.

6.25.1 Detailed Description

Declares the classes for holding row data from a result set.

6.26 sql_buffer.h File Reference

Declares the SQLBuffer class.

```
#include "refcounted.h"
#include "type_info.h"
#include <string>
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::SQLBuffer**

  Holds SQL data in string form plus type information for use in converting the string to compatible C++ data types.

Typedefs

- typedef RefCountedPointer< SQLBuffer > **mysqlpp::RefCountedBuffer**

  Reference-counted version of ***SQLBuffer*** *(p. 189).*

6.26.1 Detailed Description

Declares the SQLBuffer class.

6.27 sql_types.h File Reference

Declares the closest C++ equivalent of each MySQL column type.

`#include "common.h"`

`#include "tiny_int.h"`

`#include <string>`

`#include <stdint.h>`

Namespaces

- namespace **mysqlpp**

6.27.1 Detailed Description

Declares the closest C++ equivalent of each MySQL column type.

The typedefs defined here are only for the "non-NULL" variants. To get nullable versions, wrap the appropriate type in the `Null<T>` template. See **null.h** (p. 280) for more information.

6.28 stadapter.h File Reference

Declares the SQLTypeAdapter class.

```
#include "common.h"
#include "datetime.h"
#include "null.h"
#include "sql_buffer.h"
#include "sql_types.h"
#include <stdexcept>
#include <string>
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::SQLTypeAdapter**
  Converts many different data types to strings suitable for use in SQL queries.

6.28.1 Detailed Description

Declares the SQLTypeAdapter class.

6.29 stream2string.h File Reference

Declares an adapter that converts something that can be inserted into a C++ stream into a std::string type.

```
#include <sstream>
#include <string>
```

Namespaces

- namespace **mysqlpp**

Functions

- template<class T>
  std::string **mysqlpp::stream2string** (const T &object)

  Converts anything you can insert into a C++ stream to a `std::string` *via* `std::ostringstream`*.*

6.29.1 Detailed Description

Declares an adapter that converts something that can be inserted into a C++ stream into a std::string type.

6.30 tcp_connection.h File Reference

Declares the TCPConnection class.

```
#include "connection.h"
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::TCPConnection**

  Specialization of `Connection` (p. 32) *for TCP/IP.*

6.30.1 Detailed Description

Declares the TCPConnection class.

6.31 tiny_int.h File Reference

Declares class for holding a SQL TINYINT.

```
#include "common.h"
#include <ostream>
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::tiny_int**< **VT** >
  Class for holding an SQL `TINYINT` *value.*

Functions

- template<typename VT>
  std::ostream & **mysqlpp::operator**<< (std::ostream &os, tiny_int< VT > i)
  Insert a **`tiny_int`** `(p. 232)` *into a C++ stream.*

6.31.1 Detailed Description

Declares class for holding a SQL TINYINT.

6.32 transaction.h File Reference

Declares the Transaction class.

```
#include "common.h"
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::Transaction**

  Helper object for creating exception-safe SQL transactions.

6.32.1 Detailed Description

Declares the Transaction class.

This object works with the Connection class to automate the use of MySQL transactions. It allows you to express these transactions directly in C++ code instead of sending the raw SQL commands.

6.33 type_info.h File Reference

Declares classes that provide an interface between the SQL and C++ type systems.

```
#include "common.h"
#include "exceptions.h"
#include <map>
#include <sstream>
#include <typeinfo>
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::mysql_type_info**

  SQL field type information.

Functions

- bool **mysqlpp::operator==** (const mysql_type_info &a, const mysql_type_info &b)

  Returns true if two ***mysql_type_info*** *(*p. *102) objects are equal.*

- bool **mysqlpp::operator!=** (const mysql_type_info &a, const mysql_type_info &b)

  Returns true if two ***mysql_type_info*** *(*p. *102) objects are not equal.*

- bool **mysqlpp::operator==** (const std::type_info &a, const mysql_type_info &b)

  Returns true if a given ***mysql_type_info*** *(*p. *102) object is equal to a given C++ type_-info object.*

- bool **mysqlpp::operator!=** (const std::type_info &a, const mysql_type_info &b)

  Returns true if a given ***mysql_type_info*** *(*p. *102) object is not equal to a given C++ type_info object.*

- bool **mysqlpp::operator==** (const mysql_type_info &a, const std::type_info &b)

  Returns true if a given ***mysql_type_info*** *(*p. *102) object is equal to a given C++ type_-info object.*

- bool **mysqlpp::operator!=** (const mysql_type_info &a, const std::type_info &b)

  Returns true if a given ***mysql_type_info*** *(*p. *102) object is not equal to a given C++ type_info object.*

6.33.1 Detailed Description

Declares classes that provide an interface between the SQL and C++ type systems.

These classes are mostly used internal to the library.

6.34 uds_connection.h File Reference

Declares the UnixDomainSocketConnection class.

`#include "connection.h"`

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::UnixDomainSocketConnection**

  Specialization of **`Connection`** `(p. 32)` *for Unix domain sockets.*

6.34.1 Detailed Description

Declares the UnixDomainSocketConnection class.

6.35 vallist.h File Reference

Declares templates for holding lists of values.

```
#include "manip.h"
#include <string>
#include <vector>
```

Namespaces

- namespace **mysqlpp**

Classes

- struct **mysqlpp::equal_list_ba< Seq1, Seq2, Manip >**

  Holds two lists of items, typically used to construct a SQL "equals clause".

- struct **mysqlpp::equal_list_b< Seq1, Seq2, Manip >**

  *Same as **equal_list_ba** (p. 82), plus the option to have some elements of the equals clause suppressed.*

- struct **mysqlpp::value_list_ba< Seq, Manip >**

  Holds a list of items, typically used to construct a SQL "value list".

- struct **mysqlpp::value_list_b< Seq, Manip >**

  *Same as **value_list_ba** (p. 253), plus the option to have some elements of the list suppressed.*

Functions

- template<class Seq1, class Seq2, class Manip>
  std::ostream & **mysqlpp::operator**<< (std::ostream &o, const equal_list_ba< Seq1, Seq2, Manip > &el)

  *Inserts an **equal_list_ba** (p. 82) into an std::ostream.*

- template<class Seq1, class Seq2, class Manip>
  std::ostream & **mysqlpp::operator**<< (std::ostream &o, const equal_list_b< Seq1, Seq2, Manip > &el)

  *Same as operator<< for **equal_list_ba** (p. 82), plus the option to suppress insertion of some list items in the stream.*

- template<class Seq, class Manip>
  std::ostream & **mysqlpp::operator**<< (std::ostream &o, const value_list_ba< Seq, Manip > &cl)

  *Inserts a **value_list_ba** (p. 253) into an std::ostream.*

- template<class Seq, class Manip>
  std::ostream & **mysqlpp::operator**<< (std::ostream &o, const value_list_b< Seq, Manip > &cl)

  *Same as operator<< for **value_list_ba** (p. 253), plus the option to suppress insertion of some list items in the stream.*

- void **mysqlpp::create_vector** (size_t size, std::vector< bool > &v, bool t0, bool t1=false, bool t2=false, bool t3=false, bool t4=false, bool t5=false, bool t6=false, bool t7=false, bool t8=false, bool t9=false, bool ta=false, bool tb=false, bool tc=false)

  Create a vector of bool with the given arguments as values.

- template<class Container>
  void **mysqlpp::create_vector** (const Container &c, std::vector< bool > &v, std::string s0, std::string s1, std::string s2, std::string s3, std::string s4, std::string s5, std::string s6, std::string s7, std::string s8, std::string s9, std::string sa, std::string sb, std::string sc)

  Create a vector of bool using a list of named fields.

- template<class Seq>
  value_list_ba< Seq, do_nothing_type0 > **mysqlpp::value_list** (const Seq &s, const char ∗d=",")

  *Constructs a **value_list_ba** (p. 253).*

- template<class Seq, class Manip>
  value_list_ba< Seq, Manip > **mysqlpp::value_list** (const Seq &s, const char ∗d, Manip m)

  *Constructs a **value_list_ba** (p. 253).*

- template<class Seq, class Manip>
  value_list_b< Seq, Manip > **mysqlpp::value_list** (const Seq &s, const char ∗d, Manip m, const std::vector< bool > &vb)

  *Constructs a **value_list_b** (p. 251) (sparse value list).*

- template<class Seq, class Manip>
  value_list_b< Seq, Manip > **mysqlpp::value_list** (const Seq &s, const char ∗d, Manip m, bool t0, bool t1=false, bool t2=false, bool t3=false, bool t4=false, bool t5=false, bool t6=false, bool t7=false, bool t8=false, bool t9=false, bool ta=false, bool tb=false, bool tc=false)

*Constructs a **value_list_b** (p. 251) (sparse value list).*

- template<class Seq>
  value_list_b< Seq, do_nothing_type0 > **mysqlpp::value_list** (const Seq &s, const char ∗d, bool t0, bool t1=false, bool t2=false, bool t3=false, bool t4=false, bool t5=false, bool t6=false, bool t7=false, bool t8=false, bool t9=false, bool ta=false, bool tb=false, bool tc=false)

  Constructs a sparse value list.

- template<class Seq>
  value_list_b< Seq, do_nothing_type0 > **mysqlpp::value_list** (const Seq &s, bool t0, bool t1=false, bool t2=false, bool t3=false, bool t4=false, bool t5=false, bool t6=false, bool t7=false, bool t8=false, bool t9=false, bool ta=false, bool tb=false, bool tc=false)

  Constructs a sparse value list.

- template<class Seq1, class Seq2>
  equal_list_ba< Seq1, Seq2, do_nothing_type0 > **mysqlpp::equal_list** (const Seq1 &s1, const Seq2 &s2, const char ∗d=",", const char ∗e=" = ")

  *Constructs an **equal_list_ba** (p. 82).*

- template<class Seq1, class Seq2, class Manip>
  equal_list_ba< Seq1, Seq2, Manip > **mysqlpp::equal_list** (const Seq1 &s1, const Seq2 &s2, const char ∗d, const char ∗e, Manip m)

  *Constructs an **equal_list_ba** (p. 82).*

- template<class Seq1, class Seq2, class Manip>
  equal_list_b< Seq1, Seq2, Manip > **mysqlpp::equal_list** (const Seq1 &s1, const Seq2 &s2, const char ∗d, const char ∗e, Manip m, const std::vector< bool > &vb)

  *Constructs a **equal_list_b** (p. 80) (sparse equal list).*

- template<class Seq1, class Seq2, class Manip>
  equal_list_b< Seq1, Seq2, Manip > **mysqlpp::equal_list** (const Seq1 &s1, const Seq2 &s2, const char ∗d, const char ∗e, Manip m, bool t0, bool t1=false, bool t2=false, bool t3=false, bool t4=false, bool t5=false, bool t6=false, bool t7=false, bool t8=false, bool t9=false, bool ta=false, bool tb=false, bool tc=false)

  *Constructs a **equal_list_b** (p. 80) (sparse equal list).*

- template<class Seq1, class Seq2>
  equal_list_b< Seq1, Seq2, do_nothing_type0 > **mysqlpp::equal_list** (const Seq1 &s1, const Seq2 &s2, const char ∗d, const char ∗e, bool t0, bool t1=false, bool t2=false, bool t3=false, bool t4=false, bool t5=false, bool t6=false,

bool t7=false, bool t8=false, bool t9=false, bool ta=false, bool tb=false, bool tc=false)

*Constructs a **equal_list_b** (p. 80) (sparse equal list).*

- template<class Seq1, class Seq2>
  equal_list_b< Seq1, Seq2, do_nothing_type0 > **mysqlpp::equal_list** (const Seq1 &s1, const Seq2 &s2, const char ∗d, bool t0, bool t1=false, bool t2=false, bool t3=false, bool t4=false, bool t5=false, bool t6=false, bool t7=false, bool t8=false, bool t9=false, bool ta=false, bool tb=false, bool tc=false)

  *Constructs a **equal_list_b** (p. 80) (sparse equal list).*

- template<class Seq1, class Seq2>
  equal_list_b< Seq1, Seq2, do_nothing_type0 > **mysqlpp::equal_list** (const Seq1 &s1, const Seq2 &s2, bool t0, bool t1=false, bool t2=false, bool t3=false, bool t4=false, bool t5=false, bool t6=false, bool t7=false, bool t8=false, bool t9=false, bool ta=false, bool tb=false, bool tc=false)

  *Constructs a **equal_list_b** (p. 80) (sparse equal list).*

6.35.1 Detailed Description

Declares templates for holding lists of values.

6.36 wnp_connection.h File Reference

Declares the WindowsNamedPipeConnection class.

```
#include "connection.h"
```

Namespaces

- namespace **mysqlpp**

Classes

- class **mysqlpp::WindowsNamedPipeConnection**

  Specialization of **`Connection`** `(p. 32)` *for Windows named pipes.*

6.36.1 Detailed Description

Declares the WindowsNamedPipeConnection class.

Index