

# MySQL++ Reference Manual

## 3.0.5

Generated by Doxygen 1.4.6

Wed Aug 6 17:19:07 2008



# Contents

<b>1</b>	<b>MySQL++ Reference Manual</b>	<b>1</b>
1.1	Getting Started . . . . .	1
1.2	Major Classes . . . . .	1
1.3	Major Files . . . . .	1
1.4	If You Have Questions... . . . .	2
1.5	Licensing . . . . .	2
<b>2</b>	<b>MySQL++ Hierarchical Index</b>	<b>3</b>
2.1	MySQL++ Class Hierarchy . . . . .	3
<b>3</b>	<b>MySQL++ Class Index</b>	<b>7</b>
3.1	MySQL++ Class List . . . . .	7
<b>4</b>	<b>MySQL++ File Index</b>	<b>11</b>
4.1	MySQL++ File List . . . . .	11
<b>5</b>	<b>MySQL++ Class Documentation</b>	<b>13</b>
5.1	AutoFlag< T > Class Template Reference . . . . .	13
5.2	mysqlpp::BadConversion Class Reference . . . . .	15
5.3	mysqlpp::BadFieldName Class Reference . . . . .	18
5.4	mysqlpp::BadOption Class Reference . . . . .	20
5.5	mysqlpp::BadParamCount Class Reference . . . . .	22
5.6	mysqlpp::BadQuery Class Reference . . . . .	23
5.7	mysqlpp::BeecryptMutex Class Reference . . . . .	26

5.8	Comparable< T > Class Template Reference . . . . .	28
5.9	mysqlpp::CompressOption Class Reference . . . . .	30
5.10	mysqlpp::Connection Class Reference . . . . .	31
5.11	mysqlpp::ConnectionFailed Class Reference . . . . .	41
5.12	mysqlpp::ConnectionPool Class Reference . . . . .	43
5.13	mysqlpp::ConnectTimeoutOption Class Reference . . . . .	47
5.14	mysqlpp::DataOption< T > Class Template Reference . . . . .	48
5.15	mysqlpp::Date Class Reference . . . . .	50
5.16	mysqlpp::DateTime Class Reference . . . . .	54
5.17	mysqlpp::DBDriver Class Reference . . . . .	59
5.18	mysqlpp::DBSelectionFailed Class Reference . . . . .	73
5.19	mysqlpp::equal_list_b< Seq1, Seq2, Manip > Struct Template Reference . . . . .	75
5.20	mysqlpp::equal_list_ba< Seq1, Seq2, Manip > Struct Template Reference . . . . .	77
5.21	mysqlpp::Exception Class Reference . . . . .	79
5.22	mysqlpp::Field Class Reference . . . . .	81
5.23	mysqlpp::FieldNames Class Reference . . . . .	84
5.24	mysqlpp::FieldTypes Class Reference . . . . .	85
5.25	mysqlpp::FoundRowsOption Class Reference . . . . .	86
5.26	mysqlpp::GuessConnectionOption Class Reference . . . . .	87
5.27	mysqlpp::IgnoreSpaceOption Class Reference . . . . .	88
5.28	mysqlpp::InitCommandOption Class Reference . . . . .	89
5.29	mysqlpp::InteractiveOption Class Reference . . . . .	90
5.30	mysqlpp::LocalFilesOption Class Reference . . . . .	91
5.31	mysqlpp::LocalInfileOption Class Reference . . . . .	92
5.32	mysqlpp::MultiResultsOption Class Reference . . . . .	93
5.33	mysqlpp::MultiStatementsOption Class Reference . . . . .	94
5.34	mysqlpp::MutexFailed Class Reference . . . . .	95
5.35	mysqlpp::mysql_type_info Class Reference . . . . .	96
5.36	mysqlpp::NamedPipeOption Class Reference . . . . .	101
5.37	mysqlpp::NoExceptions Class Reference . . . . .	102

5.38	mysqlpp::NoSchemaOption Class Reference . . . . .	104
5.39	mysqlpp::Null< Type, Behavior > Class Template Reference . . . . .	105
5.40	mysqlpp::null_type Class Reference . . . . .	110
5.41	mysqlpp::NullIsBlank Struct Reference . . . . .	111
5.42	mysqlpp::NullIsNull Struct Reference . . . . .	112
5.43	mysqlpp::NullIsZero Struct Reference . . . . .	113
5.44	mysqlpp::ObjectNotInitialized Class Reference . . . . .	114
5.45	mysqlpp::Option Class Reference . . . . .	115
5.46	mysqlpp::OptionalExceptions Class Reference . . . . .	117
5.47	mysqlpp::Query Class Reference . . . . .	119
5.48	mysqlpp::ReadDefaultFileOption Class Reference . . . . .	141
5.49	mysqlpp::ReadDefaultGroupOption Class Reference . . . . .	142
5.50	mysqlpp::ReadTimeoutOption Class Reference . . . . .	143
5.51	mysqlpp::ReconnectOption Class Reference . . . . .	144
5.52	mysqlpp::RefCountedPointer< T, Destroyer > Class Template Reference	145
5.53	mysqlpp::RefCountedPointerDestroyer< T > Struct Template Reference	150
5.54	mysqlpp::RefCountedPointerDestroyer< MYSQL_RES > Struct Template Reference . . . . .	151
5.55	mysqlpp::ReportDataTruncationOption Class Reference . . . . .	152
5.56	mysqlpp::ResultBase Class Reference . . . . .	153
5.57	mysqlpp::Row Class Reference . . . . .	157
5.58	mysqlpp::ScopedLock Class Reference . . . . .	170
5.59	mysqlpp::SecureAuthOption Class Reference . . . . .	171
5.60	mysqlpp::SelfTestFailed Class Reference . . . . .	172
5.61	mysqlpp::Set< Container > Class Template Reference . . . . .	173
5.62	mysqlpp::SetCharsetDirOption Class Reference . . . . .	174
5.63	mysqlpp::SetCharsetNameOption Class Reference . . . . .	175
5.64	mysqlpp::SetClientIpOption Class Reference . . . . .	176
5.65	mysqlpp::SharedMemoryBaseNameOption Class Reference . . . . .	177
5.66	mysqlpp::SimpleResult Class Reference . . . . .	178
5.67	mysqlpp::SQLBuffer Class Reference . . . . .	180

5.68	mysqlpp::SQLParseElement Struct Reference . . . . .	183
5.69	mysqlpp::SQLQueryParms Class Reference . . . . .	185
5.70	mysqlpp::SQLTypeAdapter Class Reference . . . . .	188
5.71	mysqlpp::SslOption Class Reference . . . . .	196
5.72	mysqlpp::StoreQueryResult Class Reference . . . . .	198
5.73	mysqlpp::String Class Reference . . . . .	200
5.74	mysqlpp::TCPConnection Class Reference . . . . .	212
5.75	mysqlpp::Time Class Reference . . . . .	216
5.76	mysqlpp::tiny_int< VT > Class Template Reference . . . . .	220
5.77	mysqlpp::TooOld< ConnInfoT > Class Template Reference . . . . .	224
5.78	mysqlpp::Transaction Class Reference . . . . .	225
5.79	mysqlpp::TypeLookupFailed Class Reference . . . . .	227
5.80	mysqlpp::UnixDomainSocketConnection Class Reference . . . . .	229
5.81	mysqlpp::UseEmbeddedConnectionOption Class Reference . . . . .	232
5.82	mysqlpp::UseQueryError Class Reference . . . . .	233
5.83	mysqlpp::UseQueryResult Class Reference . . . . .	234
5.84	mysqlpp::UseRemoteConnectionOption Class Reference . . . . .	237
5.85	mysqlpp::value_list_b< Seq, Manip > Struct Template Reference . . . . .	238
5.86	mysqlpp::value_list_ba< Seq, Manip > Struct Template Reference . . . . .	240
5.87	mysqlpp::WindowsNamedPipeConnection Class Reference . . . . .	242
5.88	mysqlpp::WriteTimeoutOption Class Reference . . . . .	245
<b>6</b>	<b>MySQL++ File Documentation</b>	<b>247</b>
6.1	autoflag.h File Reference . . . . .	247
6.2	beemutex.h File Reference . . . . .	248
6.3	common.h File Reference . . . . .	250
6.4	comparable.h File Reference . . . . .	252
6.5	connection.h File Reference . . . . .	253
6.6	cpool.h File Reference . . . . .	255
6.7	custom.h File Reference . . . . .	256
6.8	datetime.h File Reference . . . . .	257

6.9	dbdriver.h File Reference . . . . .	259
6.10	exceptions.h File Reference . . . . .	260
6.11	field.h File Reference . . . . .	263
6.12	field_names.h File Reference . . . . .	265
6.13	field_types.h File Reference . . . . .	266
6.14	manip.h File Reference . . . . .	267
6.15	myset.h File Reference . . . . .	269
6.16	mysql++.h File Reference . . . . .	271
6.17	mystring.h File Reference . . . . .	274
6.18	noexceptions.h File Reference . . . . .	277
6.19	null.h File Reference . . . . .	279
6.20	options.h File Reference . . . . .	281
6.21	qparms.h File Reference . . . . .	285
6.22	query.h File Reference . . . . .	287
6.23	refcounted.h File Reference . . . . .	290
6.24	result.h File Reference . . . . .	291
6.25	row.h File Reference . . . . .	293
6.26	sql_buffer.h File Reference . . . . .	295
6.27	sql_types.h File Reference . . . . .	297
6.28	stadapter.h File Reference . . . . .	298
6.29	stream2string.h File Reference . . . . .	300
6.30	tcp_connection.h File Reference . . . . .	301
6.31	tiny_int.h File Reference . . . . .	302
6.32	transaction.h File Reference . . . . .	304
6.33	type_info.h File Reference . . . . .	305
6.34	uds_connection.h File Reference . . . . .	307
6.35	vallist.h File Reference . . . . .	308
6.36	wnp_connection.h File Reference . . . . .	313





# 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. 31), `mysqlpp::Query`(p. 119), `mysqlpp::Row`(p. 157), `mysqlpp::StoreQueryResult`(p. 198), and `mysqlpp::UseQueryResult`(p. 234).

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. 256). (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

# MySQL++ Hierarchical Index

### 2.1 MySQL++ Class Hierarchy

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

AutoFlag< T > . . . . .	13
mysqlpp::BeecryptMutex . . . . .	26
Comparable< T > . . . . .	28
Comparable< Date > . . . . .	28
mysqlpp::Date . . . . .	50
Comparable< DateTime > . . . . .	28
mysqlpp::DateTime . . . . .	54
Comparable< Time > . . . . .	28
mysqlpp::Time . . . . .	216
mysqlpp::ConnectionPool . . . . .	43
mysqlpp::DBDriver . . . . .	59
mysqlpp::equal_list_b< Seq1, Seq2, Manip > . . . . .	75
mysqlpp::equal_list_ba< Seq1, Seq2, Manip > . . . . .	77
mysqlpp::Exception . . . . .	79
mysqlpp::BadConversion . . . . .	15
mysqlpp::BadFieldName . . . . .	18
mysqlpp::BadOption . . . . .	20
mysqlpp::BadParamCount . . . . .	22
mysqlpp::BadQuery . . . . .	23
mysqlpp::ConnectionFailed . . . . .	41
mysqlpp::DBSelectionFailed . . . . .	73
mysqlpp::MutexFailed . . . . .	95
mysqlpp::ObjectNotInitialized . . . . .	114
mysqlpp::SelfTestFailed . . . . .	172

mysqlpp::TypeLookupFailed . . . . .	227
mysqlpp::UseQueryError . . . . .	233
mysqlpp::Field . . . . .	81
mysqlpp::FieldNames . . . . .	84
mysqlpp::FieldTypes . . . . .	85
mysqlpp::mysql_type_info . . . . .	96
mysqlpp::NoExceptions . . . . .	102
mysqlpp::Null< Type, Behavior > . . . . .	105
mysqlpp::null_type . . . . .	110
mysqlpp::NullIsBlank . . . . .	111
mysqlpp::NullIsNull . . . . .	112
mysqlpp::NullIsZero . . . . .	113
mysqlpp::Option . . . . .	115
mysqlpp::CompressOption . . . . .	30
mysqlpp::DataOption< T > . . . . .	48
mysqlpp::IgnoreSpaceOption . . . . .	88
mysqlpp::InitCommandOption . . . . .	89
mysqlpp::LocalFilesOption . . . . .	91
mysqlpp::MultiResultsOption . . . . .	93
mysqlpp::MultiStatementsOption . . . . .	94
mysqlpp::ReadDefaultFileOption . . . . .	141
mysqlpp::ReadDefaultGroupOption . . . . .	142
mysqlpp::SetCharsetDirOption . . . . .	174
mysqlpp::SetCharsetNameOption . . . . .	175
mysqlpp::SetClientIpOption . . . . .	176
mysqlpp::SharedMemoryBaseNameOption . . . . .	177
mysqlpp::DataOption< bool > . . . . .	48
mysqlpp::FoundRowsOption . . . . .	86
mysqlpp::InteractiveOption . . . . .	90
mysqlpp::NoSchemaOption . . . . .	104
mysqlpp::ReconnectOption . . . . .	144
mysqlpp::ReportDataTruncationOption . . . . .	152
mysqlpp::SecureAuthOption . . . . .	171
mysqlpp::DataOption< unsigned > . . . . .	48
mysqlpp::ConnectTimeoutOption . . . . .	47
mysqlpp::LocalInfileOption . . . . .	92
mysqlpp::ReadTimeoutOption . . . . .	143
mysqlpp::WriteTimeoutOption . . . . .	245
mysqlpp::GuessConnectionOption . . . . .	87
mysqlpp::NamedPipeOption . . . . .	101
mysqlpp::SslOption . . . . .	196
mysqlpp::UseEmbeddedConnectionOption . . . . .	232
mysqlpp::UseRemoteConnectionOption . . . . .	237
mysqlpp::OptionalExceptions . . . . .	117
mysqlpp::Connection . . . . .	31

mysqlpp::TCPConnection . . . . .	212
mysqlpp::UnixDomainSocketConnection . . . . .	229
mysqlpp::WindowsNamedPipeConnection . . . . .	242
mysqlpp::Query . . . . .	119
mysqlpp::ResultBase . . . . .	153
mysqlpp::StoreQueryResult . . . . .	198
mysqlpp::UseQueryResult . . . . .	234
mysqlpp::Row . . . . .	157
mysqlpp::RefCountedPointer< T, Destroyer > . . . . .	145
mysqlpp::RefCountedPointerDestroyer< T > . . . . .	150
mysqlpp::RefCountedPointerDestroyer< MYSQL_RES > . . . . .	151
mysqlpp::ScopedLock . . . . .	170
mysqlpp::Set< Container > . . . . .	173
mysqlpp::SimpleResult . . . . .	178
mysqlpp::SQLBuffer . . . . .	180
mysqlpp::SQLParseElement . . . . .	183
mysqlpp::SQLQueryParms . . . . .	185
mysqlpp::SQLTypeAdapter . . . . .	188
mysqlpp::String . . . . .	200
mysqlpp::tiny_int< VT > . . . . .	220
mysqlpp::TooOld< ConnInfoT > . . . . .	224
mysqlpp::Transaction . . . . .	225
mysqlpp::value_list_b< Seq, Manip > . . . . .	238
mysqlpp::value_list_ba< Seq, Manip > . . . . .	240



## Chapter 3

# MySQL++ Class Index

### 3.1 MySQL++ Class List

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

<b>AutoFlag</b> < <b>T</b> > (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 ) . . . . .	13
<b>mysqlpp::BadConversion</b> ( <b>Exception</b> (p. 79) thrown when a bad type conversion is attempted ) . . . . .	15
<b>mysqlpp::BadFieldName</b> ( <b>Exception</b> (p. 79) thrown when a requested named field doesn't exist ) . . . . .	18
<b>mysqlpp::BadOption</b> ( <b>Exception</b> (p. 79) thrown when you pass an unrecognized option to <b>Connection::set_option</b> (p. 39) ) . . . . .	20
<b>mysqlpp::BadParamCount</b> ( <b>Exception</b> (p. 79) thrown when not enough query parameters are provided ) . . . . .	22
<b>mysqlpp::BadQuery</b> ( <b>Exception</b> (p. 79) thrown when the database server encounters a problem while processing your query ) . . . . .	23
<b>mysqlpp::BeecryptMutex</b> (Wrapper around platform-specific mutexes ) . .	26
<b>Comparable</b> < <b>T</b> > (Mix-in that gives its subclass a full set of comparison operators ) . . . . .	28
<b>mysqlpp::CompressOption</b> (Enable data compression on the connection ) .	30
<b>mysqlpp::Connection</b> (Manages the connection to the database server ) . . .	31
<b>mysqlpp::ConnectionFailed</b> ( <b>Exception</b> (p. 79) thrown when there is a problem related to the database server connection ) . . . . .	41
<b>mysqlpp::ConnectionPool</b> (Manages a pool of connections for programs that need more than one <b>Connection</b> (p. 31) object at a time, but can't predict how many they need in advance ) . . . . .	43
<b>mysqlpp::ConnectTimeoutOption</b> (Change <b>Connection::connect</b> (p. 35) default timeout ) . . . . .	47

<b>mysqlpp::DataOption&lt; T &gt;</b> (Define abstract interface for all *Options that take a lone scalar as an argument ) . . . . .	48
<b>mysqlpp::Date</b> (C++ form of SQL's DATE type ) . . . . .	50
<b>mysqlpp::DateTime</b> (C++ form of SQL's DATETIME type ) . . . . .	54
<b>mysqlpp::DBDriver</b> (Provides a thin abstraction layer over the underlying database client library ) . . . . .	59
<b>mysqlpp::DBSelectionFailed</b> ( <b>Exception</b> (p. 79) thrown when the program tries to select a new database and the database server refuses for some reason ) . . . . .	73
<b>mysqlpp::equal_list_b&lt; Seq1, Seq2, Manip &gt;</b> (Same as <b>equal_list_ba</b> (p. 77), plus the option to have some elements of the equals clause suppressed ) . . . . .	75
<b>mysqlpp::equal_list_ba&lt; Seq1, Seq2, Manip &gt;</b> (Holds two lists of items, typically used to construct a SQL "equals clause" ) . . . . .	77
<b>mysqlpp::Exception</b> (Base class for all MySQL++ custom exceptions ) . . .	79
<b>mysqlpp::Field</b> (Class to hold information about a SQL field ) . . . . .	81
<b>mysqlpp::FieldNames</b> (Holds a list of SQL field names ) . . . . .	84
<b>mysqlpp::FieldTypes</b> (A vector of SQL field types ) . . . . .	85
<b>mysqlpp::FoundRowsOption</b> (Make <b>Query::affected_rows</b> (p. 119) return number of matched rows ) . . . . .	86
<b>mysqlpp::GuessConnectionOption</b> (Allow C API to guess what kind of connection to use ) . . . . .	87
<b>mysqlpp::IgnoreSpaceOption</b> (Allow spaces after function names in queries ) . . . . .	88
<b>mysqlpp::InitCommandOption</b> (Give SQL executed on connect ) . . . . .	89
<b>mysqlpp::InteractiveOption</b> (Assert that this is an interactive program ) . .	90
<b>mysqlpp::LocalFilesOption</b> (Enable LOAD DATA LOCAL statement ) . . .	91
<b>mysqlpp::LocalInfileOption</b> (Enable LOAD LOCAL INFILE statement ) . .	92
<b>mysqlpp::MultiResultsOption</b> (Enable multiple result sets in a reply ) . . .	93
<b>mysqlpp::MultiStatementsOption</b> (Enable multiple queries in a request to the server ) . . . . .	94
<b>mysqlpp::MutexFailed</b> ( <b>Exception</b> (p. 79) thrown when a <b>Beecrypt-Mutex</b> (p. 26) object fails ) . . . . .	95
<b>mysqlpp::mysql_type_info</b> (SQL field type information ) . . . . .	96
<b>mysqlpp::NamedPipeOption</b> (Suggest use of named pipes ) . . . . .	101
<b>mysqlpp::NoExceptions</b> (Disable exceptions in an object derived from <b>OptionalExceptions</b> (p. 117) ) . . . . .	102
<b>mysqlpp::NoSchemaOption</b> (Disable db.tbl.col syntax in queries ) . . . . .	104
<b>mysqlpp::Null&lt; Type, Behavior &gt;</b> (Class for holding data from a SQL column with the NULL attribute ) . . . . .	105
<b>mysqlpp::null_type</b> (The type of the global mysqlpp::null object ) . . . . .	110
<b>mysqlpp::NullIsBlank</b> (Class for objects that define SQL null as a blank C string ) . . . . .	111
<b>mysqlpp::NullIsNull</b> (Class for objects that define SQL null in terms of MySQL++'s <b>null_type</b> (p. 110) ) . . . . .	112
<b>mysqlpp::NullIsZero</b> (Class for objects that define SQL null as 0 ) . . . . .	113



<b>mysqlpp::ObjectNotInitialized</b> (Exception(p. 79) thrown when you try to use an object that isn't completely initialized ) . . . . .	114
<b>mysqlpp::Option</b> (Define abstract interface for all *Option subclasses ) . . . . .	115
<b>mysqlpp::OptionalExceptions</b> (Interface allowing a class to have optional exceptions ) . . . . .	117
<b>mysqlpp::Query</b> (A class for building and executing SQL queries ) . . . . .	119
<b>mysqlpp::ReadDefaultFileOption</b> (Override use of my.cnf ) . . . . .	141
<b>mysqlpp::ReadDefaultGroupOption</b> (Override use of my.cnf ) . . . . .	142
<b>mysqlpp::ReadTimeoutOption</b> (Set(p. 173) timeout for IPC data reads ) . . . . .	143
<b>mysqlpp::ReconnectOption</b> (Enable automatic reconnection to server ) . . . . .	144
<b>mysqlpp::RefCountedPointer&lt; T, Destroyer &gt;</b> (Creates an object that acts as a reference-counted pointer to another object ) . . . . .	145
<b>mysqlpp::RefCountedPointerDestroyer&lt; T &gt;</b> (Functor to call delete on the pointer you pass to it ) . . . . .	150
<b>mysqlpp::RefCountedPointerDestroyer&lt; MYSQL_RES &gt;</b> (Functor to call mysql_free_result() on the pointer you pass to it ) . . . . .	151
<b>mysqlpp::ReportDataTruncationOption</b> (Set(p. 173) reporting of data truncation errors ) . . . . .	152
<b>mysqlpp::ResultBase</b> (Base class for <b>StoreQueryResult</b> (p. 198) and <b>UseQueryResult</b> (p. 234) ) . . . . .	153
<b>mysqlpp::Row</b> (Manages rows from a result set ) . . . . .	157
<b>mysqlpp::ScopedLock</b> (Wrapper around <b>BeecryptMutex</b> (p. 26) to add scope-bound locking and unlocking ) . . . . .	170
<b>mysqlpp::SecureAuthOption</b> (Enforce use of secure authentication, refusing connection if not available ) . . . . .	171
<b>mysqlpp::SelfTestFailed</b> (Used within MySQL++'s test harness only ) . . . . .	172
<b>mysqlpp::Set&lt; Container &gt;</b> (A special std::set derivative for holding MySQL data sets ) . . . . .	173
<b>mysqlpp::SetCharsetDirOption</b> (Give path to charset definition files ) . . . . .	174
<b>mysqlpp::SetCharsetNameOption</b> (Give name of default charset ) . . . . .	175
<b>mysqlpp::SetClientIpOption</b> (Fake client IP address when connecting to embedded server ) . . . . .	176
<b>mysqlpp::SharedMemoryBaseNameOption</b> (Set(p. 173) name of shm segment for IPC ) . . . . .	177
<b>mysqlpp::SimpleResult</b> (Holds information about the result of queries that don't return rows ) . . . . .	178
<b>mysqlpp::SQLBuffer</b> (Holds SQL data in string form plus type information for use in converting the string to compatible C++ data types ) . . . . .	180
<b>mysqlpp::SQLParseElement</b> (Used within <b>Query</b> (p. 119) to hold elements for parameterized queries ) . . . . .	183
<b>mysqlpp::SQLQueryParms</b> (This class holds the parameter values for filling template queries ) . . . . .	185
<b>mysqlpp::SQLTypeAdapter</b> (Converts many different data types to strings suitable for use in SQL queries ) . . . . .	188
<b>mysqlpp::SslOption</b> (Specialized option for handling SSL parameters ) . . . . .	196

<b>mysqlpp::StoreQueryResult</b> ( <b>StoreQueryResult</b> (p.198) set type for "store" queries ) . . . . .	198
<b>mysqlpp::String</b> (A std::string work-alike that can convert itself from SQL text data formats to C++ data types ) . . . . .	200
<b>mysqlpp::TCPConnection</b> (Specialization of <b>Connection</b> (p.31) for TCP/IP ) . . . . .	212
<b>mysqlpp::Time</b> (C++ form of SQL's TIME type ) . . . . .	216
<b>mysqlpp::tiny_int&lt; VT &gt;</b> (Class for holding an SQL TINYINT value ) . . . . .	220
<b>mysqlpp::TooOld&lt; ConnInfoT &gt;</b> (Functor to test whether a given ConnectionInfo object is "too old" ) . . . . .	224
<b>mysqlpp::Transaction</b> (Helper object for creating exception-safe SQL transactions ) . . . . .	225
<b>mysqlpp::TypeLookupFailed</b> (Thrown from the C++ to SQL data type conversion routine when it can't figure out how to map the type ) . . . . .	227
<b>mysqlpp::UnixDomainSocketConnection</b> (Specialization of <b>Connection</b> (p.31) for Unix domain sockets ) . . . . .	229
<b>mysqlpp::UseEmbeddedConnectionOption</b> (Connect to embedded server in preference to remote server ) . . . . .	232
<b>mysqlpp::UseQueryError</b> ( <b>Exception</b> (p.79) thrown when something goes wrong in processing a "use" query ) . . . . .	233
<b>mysqlpp::UseQueryResult</b> ( <b>StoreQueryResult</b> (p.198) set type for "use" queries ) . . . . .	234
<b>mysqlpp::UseRemoteConnectionOption</b> (Connect to remote server in preference to embedded server ) . . . . .	237
<b>mysqlpp::value_list_b&lt; Seq, Manip &gt;</b> (Same as <b>value_list_ba</b> (p.240), plus the option to have some elements of the list suppressed ) . . . . .	238
<b>mysqlpp::value_list_ba&lt; Seq, Manip &gt;</b> (Holds a list of items, typically used to construct a SQL "value list" ) . . . . .	240
<b>mysqlpp::WindowsNamedPipeConnection</b> (Specialization of <b>Connection</b> (p.31) for Windows named pipes ) . . . . .	242
<b>mysqlpp::WriteTimeoutOption</b> ( <b>Set</b> (p.173) timeout for IPC data reads ) . . . . .	245

## Chapter 4

# MySQL++ File Index

### 4.1 MySQL++ File List

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

<b>autoflag.h</b> (Defines a template for setting a flag within a given variable scope, and resetting it when exiting that scope ) . . . . .	247
<b>beemutex.h</b> (MUTually EXclusive lock class ) . . . . .	248
<b>common.h</b> (This file includes top-level definitions for use both internal to the library, and outside it. Contrast mysql++.h ) . . . . .	250
<b>comparable.h</b> (Declares the Comparable<T> mixin ) . . . . .	252
<b>connection.h</b> (Declares the Connection class ) . . . . .	253
<b>cpool.h</b> (Declares the ConnectionPool class ) . . . . .	255
<b>custom.h</b> (Backwards-compatibility header; loads ssqsls.h ) . . . . .	256
<b>datetime.h</b> (Declares classes to add SQL-compatible date and time types to C++'s type system ) . . . . .	257
<b>dbdriver.h</b> (Declares the DBDriver class ) . . . . .	259
<b>exceptions.h</b> (Declares the MySQL++-specific exception classes ) . . . . .	260
<b>field.h</b> (Declares the Field and Fields classes ) . . . . .	263
<b>field_names.h</b> (Declares a class to hold a list of field names ) . . . . .	265
<b>field_types.h</b> (Declares a class to hold a list of SQL field type info ) . . . . .	266
<b>manip.h</b> (Declares the Query stream manipulators and operators ) . . . . .	267
<b>myset.h</b> (Declares templates for generating custom containers used elsewhere in the library ) . . . . .	269
<b>mysql++.h</b> (The main MySQL++ header file ) . . . . .	271
<b>mystring.h</b> (Declares String class, MySQL++'s generic std::string-like class, used for holding data received from the database server ) . . . . .	274
<b>noexceptions.h</b> (Declares interface that allows exceptions to be optional ) . . . . .	277
<b>null.h</b> (Declares classes that implement SQL "null" semantics within C++'s type system ) . . . . .	279

<b>options.h</b> (Declares the Option class hierarchy, used to implement connection options in Connection and DBDriver classes ) . . . . .	281
<b>qparms.h</b> (Declares the template query parameter-related stuff ) . . . . .	285
<b>query.h</b> (Defines a class for building and executing SQL queries ) . . . . .	287
<b>refcounted.h</b> (Declares the RefCountedPointer template ) . . . . .	290
<b>result.h</b> (Declares classes for holding information about SQL query results ) .	291
<b>row.h</b> (Declares the classes for holding row data from a result set ) . . . . .	293
<b>sql_buffer.h</b> (Declares the SQLBuffer class ) . . . . .	295
<b>sql_types.h</b> (Declares the closest C++ equivalent of each MySQL column type ) . . . . .	297
<b>stadapter.h</b> (Declares the SQLTypeAdapter class ) . . . . .	298
<b>stream2string.h</b> (Declares an adapter that converts something that can be inserted into a C++ stream into a std::string type ) . . . . .	300
<b>tcp_connection.h</b> (Declares the TCPConnection class ) . . . . .	301
<b>tiny_int.h</b> (Declares class for holding a SQL TINYINT ) . . . . .	302
<b>transaction.h</b> (Declares the Transaction class ) . . . . .	304
<b>type_info.h</b> (Declares classes that provide an interface between the SQL and C++ type systems ) . . . . .	305
<b>uds_connection.h</b> (Declares the UnixDomainSocketConnection class ) . . .	307
<b>vallist.h</b> (Declares templates for holding lists of values ) . . . . .	308
<b>wnp_connection.h</b> (Declares the WindowsNamedPipeConnection class ) . .	313

## Chapter 5

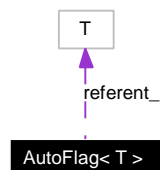
# MySQL++ Class Documentation

### 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:

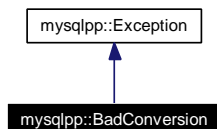
- **autoflag.h**

## 5.2 mysqlpp::BadConversion Class Reference

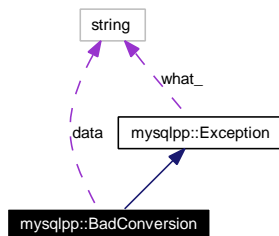
**Exception**(p. 79) thrown when a bad type conversion is attempted.

```
#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. 79) 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

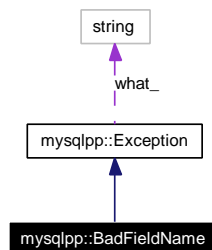
**Exception**(p. 79) thrown when a requested named field doesn't exist.

```
#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. 79) 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

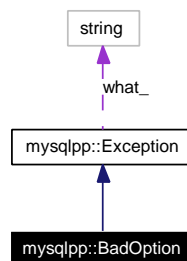
**Exception**(p. 79) thrown when you pass an unrecognized option to **Connection::set\_option()**(p. 39).

```
#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. 79) thrown when you pass an unrecognized option to **Connection::set\_option()**(p. 39).

## 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 **Bad-Option**(p. 20) exceptions programmatically.

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

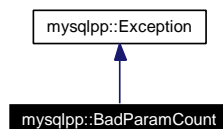
- **exceptions.h**

## 5.5 mysqlpp::BadParamCount Class Reference

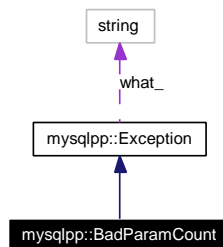
**Exception**(p. 79) thrown when not enough query parameters are provided.

```
#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. 79) 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:

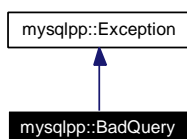
- **exceptions.h**

## 5.6 mysqlpp::BadQuery Class Reference

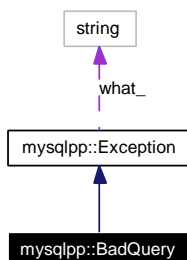
**Exception**(p. 79) thrown when the database server encounters a problem while processing your query.

```
#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. 80).*

#### 5.6.1 Detailed Description

**Exception**(p. 79) 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. 32)'s return value at the point the exception is thrown. We do this because when using the **Transaction**(p. 225) 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. 32) 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. 79) 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. 80).

This may return the same value as **Connection::errnum()**(p. 32), 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. 95) 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:

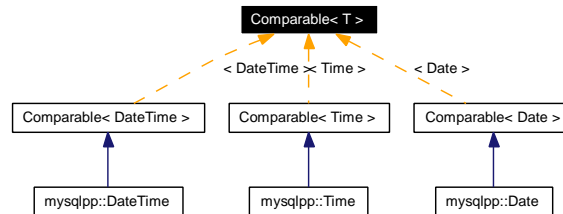
- **beemutex.h**
- beemutex.cpp

## 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 if they are equal, and > 0 if this object is "after" the other.

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

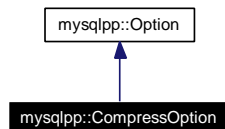
- **comparable.h**

## 5.9 mysqlpp::CompressOption Class Reference

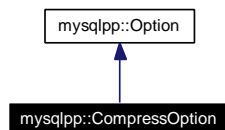
Enable data compression on the connection.

```
#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:

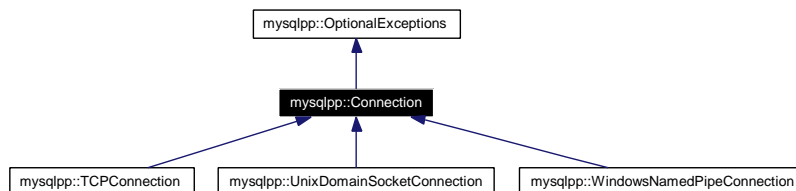
- **options.h**

## 5.10 mysqlpp::Connection Class Reference

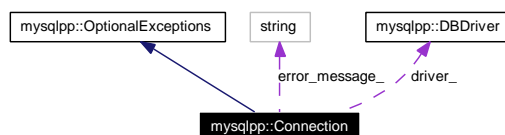
Manages the connection to the database server.

```
#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. 31) 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. 64).*

## 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. 59), 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.

### 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. 31) object

## 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.

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

Returns true if connection was established successfully.

**Returns:**

true if connection was established successfully

#### 5.10.3.3 **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

#### 5.10.3.4 **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.

#### 5.10.3.5 **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

**5.10.3.6 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

**5.10.3.7 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.

**5.10.3.8 std::string mysqlpp::Connection::ipc\_info () const**

Get information about the IPC connection to the database server.

**String**(p. 200) contains info about type of connection (e.g. TCP/IP, named pipe, Unix socket...) and the server hostname.

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

Kill a database server thread.

**See also:**

**thread\_id()**(p. 39)

**5.10.3.10 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:

///

Prior to MySQL++ v3, the object was always `false` 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. 36) instead.

#### 5.10.3.11 `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.

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

Return a new query object.

**Parameters:**

*qstr* initial query string

#### 5.10.3.13 `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. 128) 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. 119) object

#### 5.10.3.14 `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

### 5.10.3.15 bool mysqlpp::Connection::set\_option (Option \* o)

Sets a connection option.

**Parameters:**

*o* pointer to any derivative of **Option**(p. 115) allocated on the heap

Objects passed to this method and successfully set will be released when this **Connection**(p. 31) object is destroyed. If an error occurs while setting the option the object will be deleted immediately.

Because there are so many **Option**(p. 115) 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

### 5.10.3.16 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. 37).

### 5.10.3.17 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. 43), 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

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

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

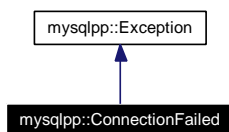


## 5.11 mysqlpp::ConnectionFailed Class Reference

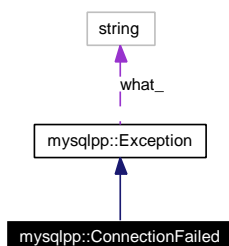
**Exception**(p. 79) thrown when there is a problem related to the database server connection.

```
#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. 80), if any.*

#### 5.11.1 Detailed Description

**Exception**(p. 79) 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. 80), 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:

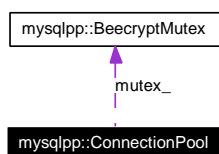
- `exceptions.h`

## 5.12 mysqlpp::ConnectionPool Class Reference

Manages a pool of connections for programs that need more than one **Connection**(p. 31) 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. 31) 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. 31) objects due to a hard limitation of the underlying C API. **Connection**(p. 31) 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. 45) in its dtor as it should.

### 5.12.3 Member Function Documentation

#### 5.12.3.1 **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. 31) 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. 45) method.

**Parameters:**

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

#### 5.12.3.2 **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. 43) what server to connect to, what login parameters to use, what connection options to enable, etc. **ConnectionPool**(p. 43) can't know any of this without your help.

**Return values:**

A connected **Connection**(p. 31) object

#### 5.12.3.3 **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. 45). Because we can't know what the derived class did to create the connection we can't reliably know how to destroy it.

#### 5.12.3.4 `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

#### 5.12.3.5 `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

#### 5.12.3.6 `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.

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

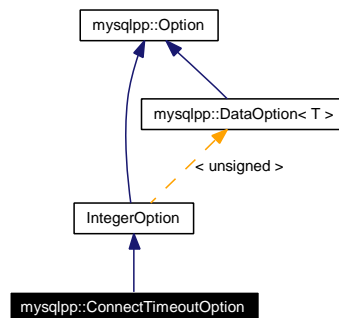
- `cpool.h`
- `cpool.cpp`

## 5.13 mysqlpp::ConnectTimeoutOption Class Reference

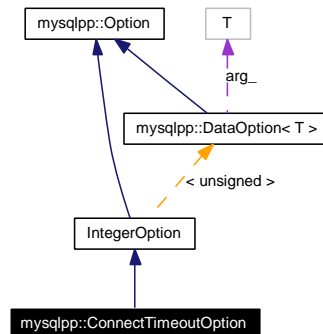
Change **Connection::connect()**(p. 35) default timeout.

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

Inheritance diagram for mysqlpp::ConnectTimeoutOption:



Collaboration diagram for mysqlpp::ConnectTimeoutOption:



### 5.13.1 Detailed Description

Change **Connection::connect()**(p. 35) default timeout.

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

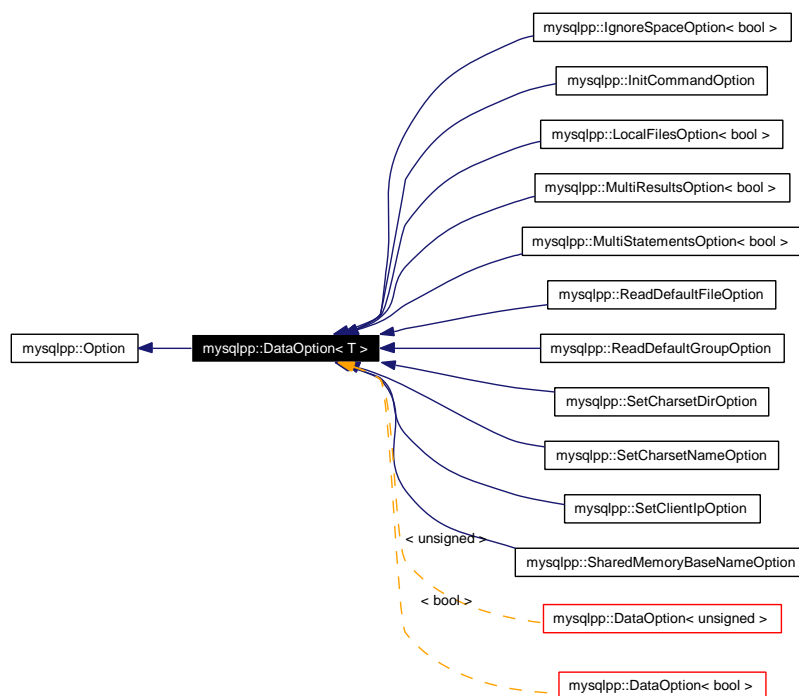
- `options.h`

## 5.14 mysqlpp::DataOption< T > Class Template Reference

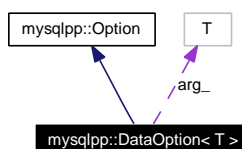
Define abstract interface for all \*Options that take a lone scalar as an argument.

```
#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:

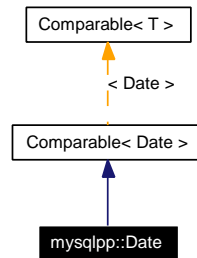
- **options.h**

## 5.15 mysqlpp::Date Class Reference

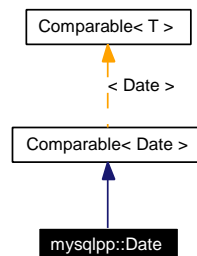
C++ form of SQL's DATE type.

```
#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. 50).*
- **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, 200)` 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, 200)`...

See also:

`Date(const char*)(p, 52)`

#### 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, 54)` 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` if they are equal, and `> 0` if this date is after the other.

### 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 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.

### 5.15.3.4 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.

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

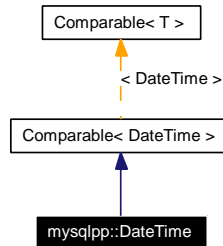
- **datetime.h**
- datetime.cpp

## 5.16 mysqlpp::DateTime Class Reference

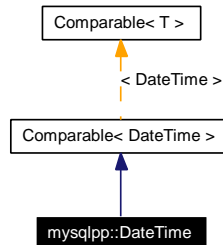
C++ form of SQL's DATETIME type.

```
#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. 50).*
- **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. 200) 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. 200)...

**See also:**

`DateTime(const char*)(p. 57)`

**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 if they are equal, and > 0 if this object is after the other.

**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 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.

**5.16.3.4 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.

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. 68).*

### 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 particular 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. 173) MySQL C API connection option.*
- `bool set_option (unsigned int option, bool arg)`  
*Set(p. 173) MySQL C API connection option.*
- `std::string set_option_default (Option *o)`  
*Same as set\_option()(p. 70), 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. 66)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 intended for use by MySQL++ users are in **Connection**(p. 31), **Query**(p. 119), **Result**, and **ResUse**, all of which delegate the actual database communication to an object of this type, created by **Connection**(p. 31). If you really need access to the low-level database driver, get it via **Connection::driver**(p. 32); don't create **DBDriver**(p. 59) 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. 68).

##### 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. 59) object

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

**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.

**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.

**5.17.4.3** `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.

**5.17.4.4** `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

**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. 64) call and no **disconnect**(p. 65). Call **ping**(p. 69) to actually test the connection's viability.



**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

**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 particular 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.

**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.

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

Return last MySQL error number associated with this connection.

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

**5.17.4.13** `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.

**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.

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

**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.

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

**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.

**5.17.4.17** `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. 67) followed by **fetch\_field()**(p. 67) 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.)

**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.

**5.17.4.19** `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.

**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.

**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.

**5.17.4.22** `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.

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

Get information about the IPC connection to the database server.

**String**(p. 200) 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.

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

Kill a MySQL server thread.

**See also:**

`thread_id()`(p. 71)

**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.

**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.

**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.

**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.

**5.17.4.31** `string mysqlpp::DBDriver::query_info ()`

Returns information about the last executed query.

Wraps `mysql_info()` in the MySQL C API

**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_status()` [inline]

Returns the database server's status.

**String**(p. 200) 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.

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

Get the database server's version number.

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

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

**Set**(p. 173) 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 `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. 39) calls. There are several private overloads that actually implement the option setting.

See also:

**Connection::set\_option(Option\*)(p. 39)** for commentary

**5.17.4.38 bool mysqlpp::DBDriver::shutdown ()**

Ask database server to shut down.

User must have the "shutdown" privilege.

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

**5.17.4.39 MYSQL\_RES\* mysqlpp::DBDriver::store\_result () [inline]**

Saves the results of the query just `execute()`(p. 66)d in memory and returns a pointer to the MySQL C API data structure the results are stored in.

**See also:**

`use_result()`(p. 72)

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

**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.

**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.

**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. 68).

**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. 59) 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.

**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. 71)

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

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

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

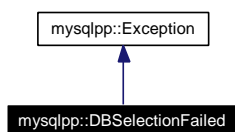


## 5.18 mysqlpp::DBSelectionFailed Class Reference

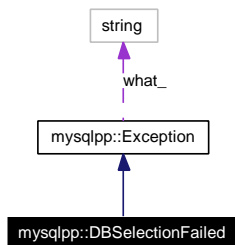
**Exception**(p. 79) thrown when the program tries to select a new database and the database server refuses for some reason.

```
#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. 80), if any.*

#### 5.18.1 Detailed Description

**Exception**(p. 79) 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. 80), 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:

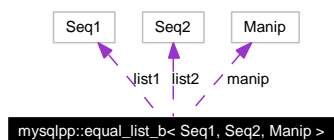
- `exceptions.h`

## 5.19 mysqlpp::equal\_list\_b< Seq1, Seq2, Manip > Struct Template Reference

Same as **equal\_list\_ba**(p.77), 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 \* **equal**  
*"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.77), 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:

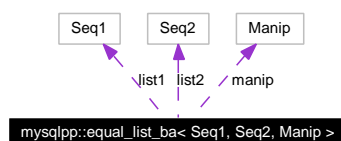
- **vallist.h**

## 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 \* **equal**  
*"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 SSQSL mechanism calls those functions when building SQL queries; you can call them yourself to do similar work. The "Harnessing SSQSL Internals" section of the user manual has some examples of this.

See also:

`equal_list_b(p. 75)`

### 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:

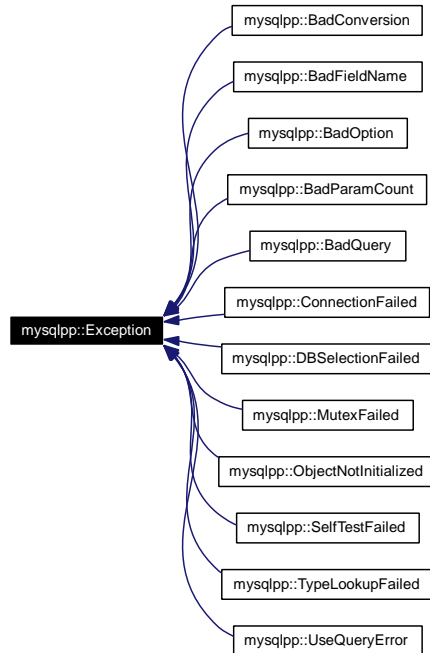
- **vallist.h**

## 5.21 mysqlpp::Exception Class Reference

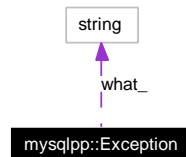
Base class for all MySQL++ custom exceptions.

```
#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**

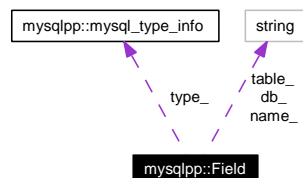


## 5.22 mysqlpp::Field Class Reference

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. 81).*
- 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:

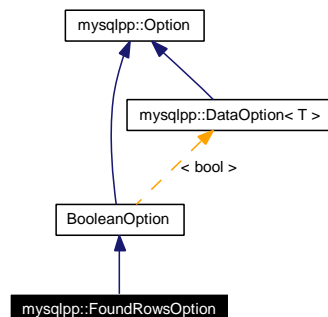
- **field\_types.h**
- **field\_types.cpp**

## 5.25 mysqlpp::FoundRowsOption Class Reference

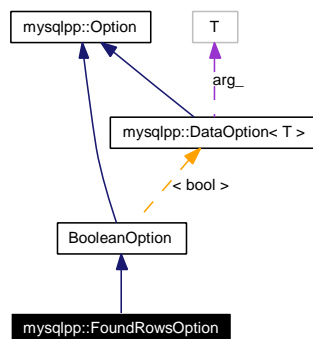
Make **Query::affected\_rows()**(p. 119) return number of matched rows.

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

Inheritance diagram for mysqlpp::FoundRowsOption:



Collaboration diagram for mysqlpp::FoundRowsOption:



### 5.25.1 Detailed Description

Make **Query::affected\_rows()**(p. 119) 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. 232) and **UseEmbeddedConnectionOption**(p. 232).

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

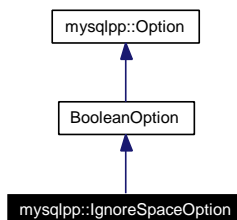
- **options.h**

## 5.27 mysqlpp::IgnoreSpaceOption Class Reference

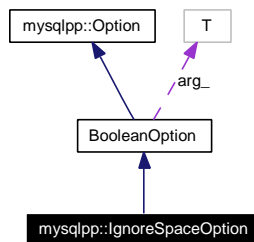
Allow spaces after function names in queries.

```
#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**

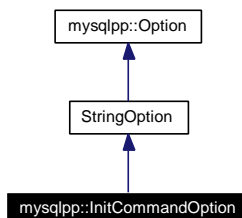


## 5.28 mysqlpp::InitCommandOption Class Reference

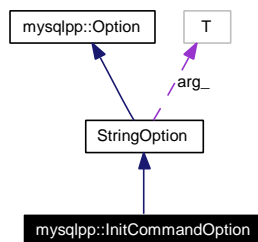
Give SQL executed on connect.

```
#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:

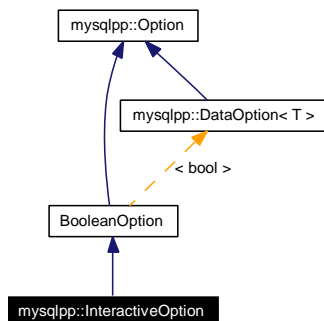
- **options.h**

## 5.29 mysqlpp::InteractiveOption Class Reference

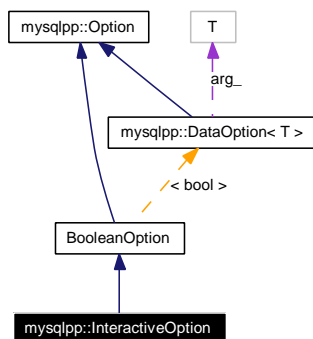
Assert that this is an interactive program.

```
#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:

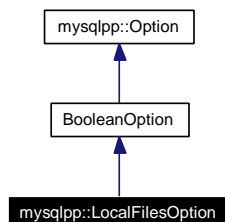
- `options.h`

## 5.30 mysqlpp::LocalFilesOption Class Reference

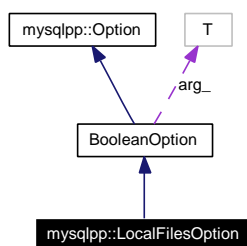
Enable LOAD DATA LOCAL statement.

```
#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:

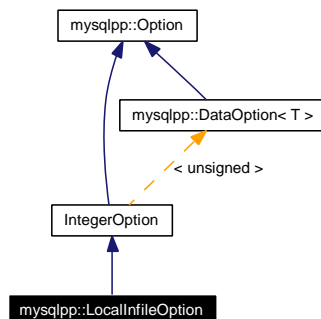
- **options.h**

## 5.31 mysqlpp::LocalInfileOption Class Reference

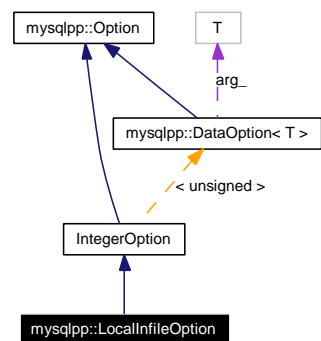
Enable LOAD LOCAL INFILE statement.

```
#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:

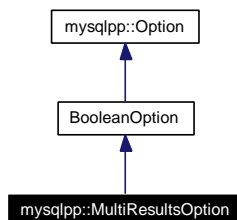
- `options.h`

## 5.32 mysqlpp::MultiResultsOption Class Reference

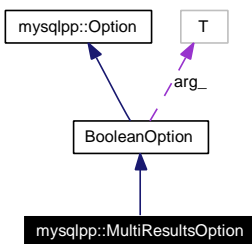
Enable multiple result sets in a reply.

```
#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:

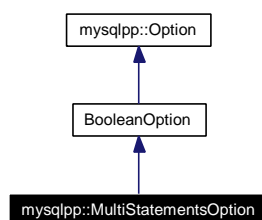
- **options.h**

### 5.33 mysqlpp::MultiStatementsOption Class Reference

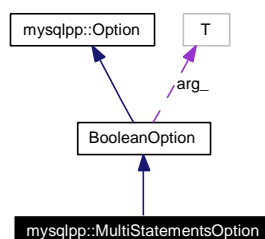
Enable multiple queries in a request to the server.

```
#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:

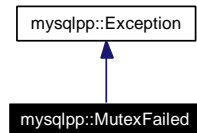
- **options.h**

## 5.34 mysqlpp::MutexFailed Class Reference

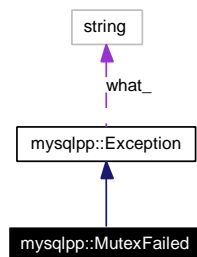
**Exception**(p. 79) thrown when a **BeecryptMutex**(p. 26) object fails.

```
#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. 79) 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. 96) 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. 105) 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. 85) 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 `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. 105)` type.

Returns the type\_info for the C++ type inside the `mysqlpp::Null(p. 105)` type. If the type is not `Null(p. 105)` then this is the same as `c_type()(p. 98)`.

### 5.35.3.2 `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.

### 5.35.3.3 `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.

### 5.35.3.4 `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.

### 5.35.3.5 `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.

#### 5.35.3.6 `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. 99) for the C++ type associated with the SQL type.

#### 5.35.3.7 `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.

#### 5.35.3.8 `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.

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

Returns the name of the SQL type.

Returns the SQL name for the type.

### 5.35.4 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.

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

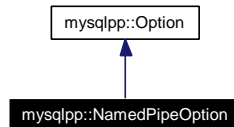
- **type\_info.h**
- type\_info.cpp

## 5.36 mysqlpp::NamedPipeOption Class Reference

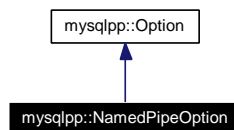
Suggest use of named pipes.

```
#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:

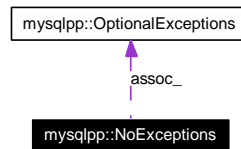
- **options.h**

## 5.37 mysqlpp::NoExceptions Class Reference

Disable exceptions in an object derived from **OptionalExceptions**(p. 117).

```
#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. 117).

This class was designed to be created on the stack, taking a reference to a subclass of **OptionalExceptions**(p. 117). (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. 117) 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:

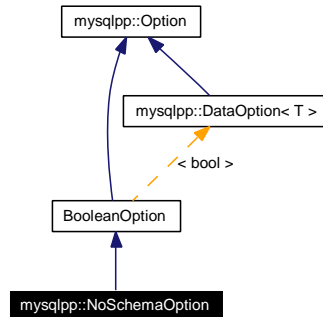
- **noexceptions.h**

## 5.38 mysqlpp::NoSchemaOption Class Reference

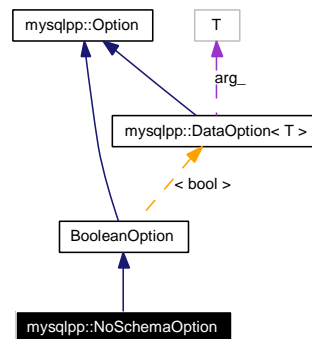
Disable db.tbl.col syntax in queries.

```
#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`

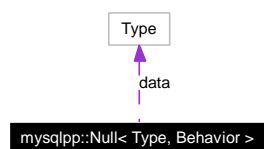


## 5.39 mysqlpp::Null< Type, Behavior > Class Template Reference

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. 105) 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. 105) 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...

```
///
```

...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. 112), `NullIsZero`(p. 113) and `NullIsBlank`(p. 111).

Otherwise, just returns the 'data' member.

**5.39.3.2** `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.

**5.39.3.3** `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.

**5.39.3.4** `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.

**5.39.3.5** `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.

**5.39.3.6** `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.

**5.39.3.7** `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.

## 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.

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. 105) 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. 105)

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. 110).

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

### 5.42.1 Detailed Description

Class for objects that define SQL null in terms of MySQL++'s **null\_type**(p. 110).

Returns a **null\_type**(p. 110) 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. 105)

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. 105)

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

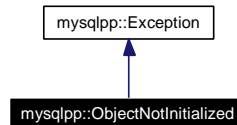
- **null.h**

## 5.44 mysqlpp::ObjectNotInitialized Class Reference

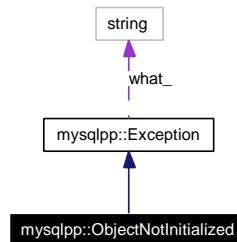
**Exception**(p. 79) thrown when you try to use an object that isn't completely initialized.

```
#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. 79) thrown when you try to use an object that isn't completely initialized.

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

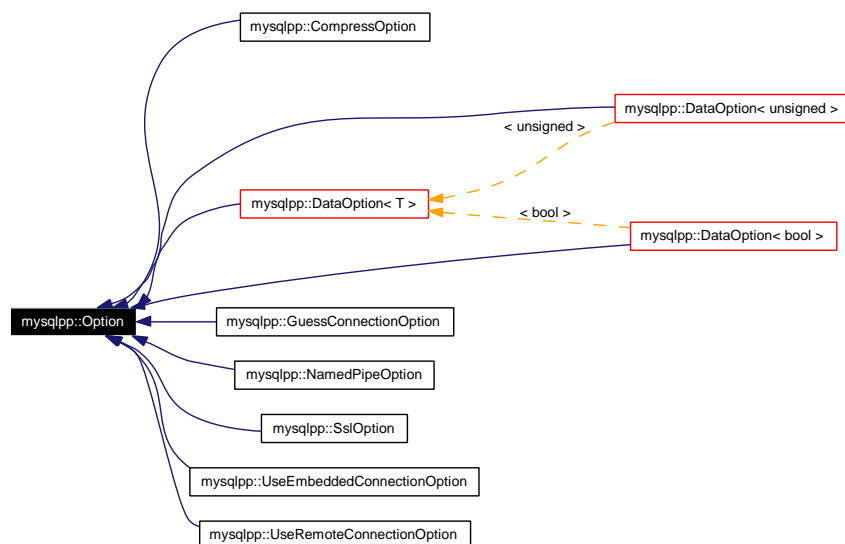
- **exceptions.h**

## 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:

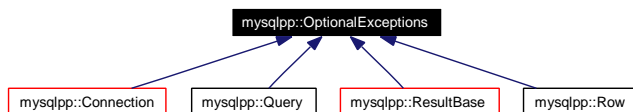
- **options.h**

## 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. 102) 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. 117) 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.

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

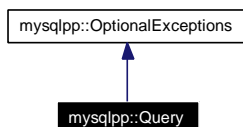
- **noexceptions.h**

## 5.47 mysqlpp::Query Class Reference

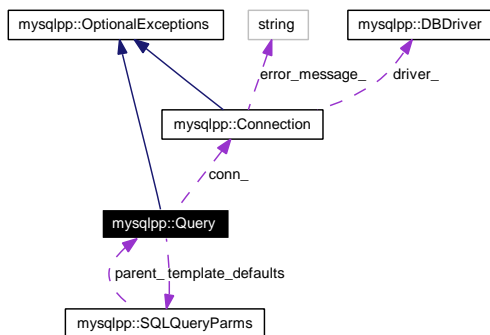
A class for building and executing SQL queries.

```
#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. 136) for std::vector.*
- `template<class T> void storein (std::deque< T > &con, const SQLTypeAdapter &s)`  
*Specialization of storein\_sequence()(p. 136) for std::deque.*
- `template<class T> void storein (std::list< T > &con, const SQLTypeAdapter &s)`  
*Specialization of storein\_sequence()(p. 136) for std::list.*
- `template<class T> void storein (std::set< T > &con, const SQLTypeAdapter &s)`  
*Specialization of storein\_set()(p. 137) for std::set.*
- `template<class T> void storein (std::multiset< T > &con, const SQLTypeAdapter &s)`  
*Specialization of storein\_set()(p. 137) 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. 119) objects directly. Instead, call **mysqlpp::Connection::query**(p. 38) 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. 128) **store\***(p. 133) 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::ostream**, 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. 128) **store\***(p. 133) 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. 119) has several special functions for generating common SQL queries from those structures. For instance, it offers the **insert**(p. 130) 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. 131) method to tell the **Query**(p. 119) object that the query string contains placeholders. Having done that, you call one of the many **exec\***(p. 128) **store\***(p. 132) or **use**(p. 139) overloads that take **SQLTypeAdapter**(p. 188) 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. 119) 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. 38).

#### 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. 38) to an empty **Query**(p. 119) 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 int mysqlpp::Query::errnum () const

Get the last error number that was set.

This just delegates to **Connection::errnum()**(p. 32). **Query**(p. 119) has nothing extra to say, so use either, as makes sense in your program.

### 5.47.3.2 const char \* mysqlpp::Query::error () const

Get the last error message that was set.

This just delegates to **Connection::error()**(p. 37). **Query**(p. 119) has nothing extra to say, so use either, as makes sense in your program.

### 5.47.3.3 `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. 119) because proper SQL escaping takes the database's current character set into account, which requires access to the **Connection**(p. 31) object the query will go out on. Also, this function is very important to MySQL++'s **Query**(p. 119) stream manipulator mechanism, so it's more convenient for this method to live in **Query**(p. 119) rather than **Connection**(p. 31).

### 5.47.3.4 `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.

#### 5.47.3.5 bool mysqlpp::Query::exec (const std::string & str)

Execute a query.

Same as **execute()**(p. 128), 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. 128), **store()**(p. 133), **storein()**(p. 135), and **use()**(p. 139)

#### 5.47.3.6 bool mysqlpp::Query::exec () [inline]

Execute a built-up query.

Same as **exec()**(p. 127), 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. 127), **execute()**(p. 128), **store()**(p. 133), **storein()**(p. 135), and **use()**(p. 139)

#### 5.47.3.7 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.

#### 5.47.3.8 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. 188) can be initialized from either a C string or a C++ string, this overload accepts query strings in either form. Beware, **SQLTypeAdapter**(p. 188) also accepts many other data types (this is its *raison d'être*), 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. 188) object than the previous one. See the template query overview above for more about this topic.

#### 5.47.3.9 SimpleResult mysqlpp::Query::execute (SQLQueryParms & *p*)

Execute template query using given parameters.

##### Parameters:

*p* parameters to use in the template query.

#### 5.47.3.10 SimpleResult mysqlpp::Query::execute () [inline]

Execute built-up query.

Use one of the **execute()**(p. 128) overloads if you don't expect the server to return a result set. For instance, a DELETE query. The returned **SimpleResult**(p. 178) 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. 128) simply executes the query that you have built up in the object in some way. (For instance, via the **insert()**(p. 130) method, or by using the object's stream interface.)



**Returns:**

**SimpleResult**(p. 178) status information about the query

**See also:**

**exec()**(p. 127), **store()**(p. 133), **storein()**(p. 135), and **use()**(p. 139)

**5.47.3.11** `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. 129), 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

**5.47.3.12** `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. 129), but it uses the query string held by the **Query**(p. 119) object already

**Parameters:**

*fn* the functor called for each row

**Returns:**

a copy of the passed functor

**5.47.3.13** `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. 129) 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

**5.47.3.14** `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. 132), `update()`(p. 138)

**5.47.3.15** `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. 132), `update()`(p. 138)

**5.47.3.16** `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.

### 5.47.3.17 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. 135) 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

### 5.47.3.18 mysqlpp::Query::operator void \* () const

Test whether the object has experienced an error condition.

Allows for code constructs like this:

```
///
```

This method returns false if either the **Query**(p. 119) object or its associated **Connection**(p. 31) object has seen an error condition since the last operation.

### 5.47.3.19 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.

### 5.47.3.20 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.

#### 5.47.3.21 `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. 130), `update()`(p. 138)

#### 5.47.3.22 `void mysqlpp::Query::reset ()`

Reset the query object so that it can be reused.

As of v3.0, **Query**(p. 119) 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. 119) object set up for making template queries, then want to build queries using one of the other methods. (Static strings, SSQSL, or the stream interface.)

#### 5.47.3.23 `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. 132)` instead. **SQLTypeAdapter**(p. 188) converts from plain C strings and other useful data types implicitly.

#### 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. 188) can be initialized from either a C string or a C++ string, this overload accepts query strings in either form. Beware, **SQLTypeAdapter**(p. 188) also accepts many other data types (this is its *raison d'être*), 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. 188) object than the previous one. See the template query overview above for more about this topic.

#### 5.47.3.25 StoreQueryResult mysqlpp::Query::store (SQLQueryParms & p)

Store results from a template query using given parameters.

##### Parameters:

*p* parameters to use in the template query.

#### 5.47.3.26 StoreQueryResult mysqlpp::Query::store () [inline]

Execute a query that can return a result set.

Use one of the **store()**(p. 133) 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. 133).

You must use **store()**(p. 133), **storein()**(p. 135) 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. 127) and **execute()**(p. 128) 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. 128).

##### Returns:

**StoreQueryResult**(p. 198) object containing entire result set

##### See also:

**exec()**(p. 127), **execute()**(p. 128), **storein()**(p. 135), and **use()**(p. 139)

#### 5.47.3.27 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. 134), but it uses the query string held by the `Query`(p. 119) 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

**5.47.3.28** `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. 134), 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

**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

**5.47.3.30 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. 133) 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. 39).

Multi-queries only exist in MySQL v4.1 and higher. Therefore, this function just wraps `store()`(p. 133) when built against older API libraries.

**Returns:**

`StoreQueryResult`(p. 198) object containing the next result set.

**5.47.3.31 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. 136) or `storein_set()`(p. 137), 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. 127), **execute()**(p. 128), **store()**(p. 133), and **use()**(p. 139) for alternative query execution mechanisms.

#### 5.47.3.32 **template<class Seq> void mysqlpp::Query::storein\_sequence (Seq & con, SQLQueryParms & p) [inline]**

Execute template query using given parameters, storing.

##### Parameters:

- con* container that will receive the results
- p* parameters to use in the template query.

#### 5.47.3.33 **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. 119) 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. 188) object than the previous one. See the template query overview above for more about this topic.

#### 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. 133) 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. 128), 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. 127), `execute()`(p. 128), `store()`(p. 133), and `use()`(p. 139)

### 5.47.3.35 `template<class Set> void mysqlpp::Query::storein_set (Set & con, SQLQueryParms & p) [inline]`

Execute template query using given parameters, storing.

**Parameters:**

*con* container that will receive the results

*p* parameters to use in the template query.

### 5.47.3.36 `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. 119) 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. 188) object than the previous one. See the template query overview above for more about this topic.

### 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. 136), 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 `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

#### 5.47.3.39 `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. 188) 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. 188) 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. 188) object than the previous one. See the template query overview above for more about this topic.

#### 5.47.3.40 `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. 130), **replace**(p. 132)

#### 5.47.3.41 `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. 139) instead. **SQLTypeAdapter**(p. 188) converts from plain C strings and other useful data types implicitly.

#### 5.47.3.42 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. 188) can be initialized from either a C string or a C++ string, this overload accepts query strings in either form. Beware, **SQLTypeAdapter**(p. 188) also accepts many other data types (this is its *raison d'être*), 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. 188) object than the previous one. See the template query overview above for more about this topic.

#### 5.47.3.43 UseQueryResult mysqlpp::Query::use (SQLQueryParms & *p*)

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

**Parameters:**

*p* parameters to use in the template query.

#### 5.47.3.44 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. 133) when there are a large number of results; **store()**(p. 133) 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. 128).

**Returns:**

**UseQueryResult**(p. 234) object that can walk through result set serially

See also:

`exec()`(p. 127), `execute()`(p. 128), `store()`(p. 133) and `storein()`(p. 135)

## 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.

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

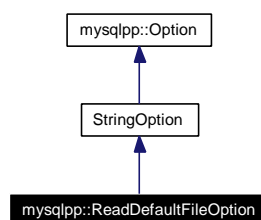
- `query.h`
- `query.cpp`

## 5.48 mysqlpp::ReadDefaultFileOption Class Reference

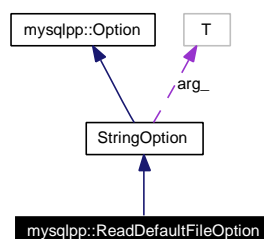
Override use of my.cnf.

```
#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:

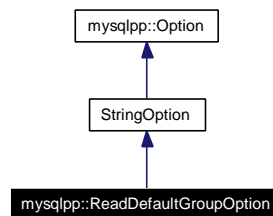
- **options.h**

## 5.49 mysqlpp::ReadDefaultGroupOption Class Reference

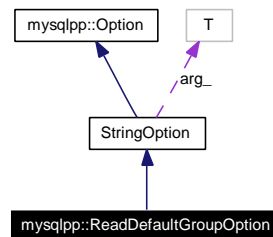
Override use of my.cnf.

```
#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:

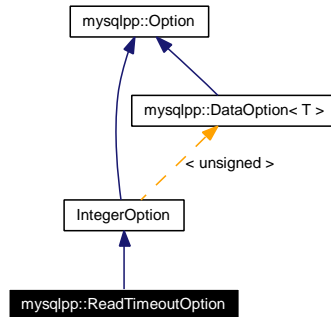
- **options.h**

## 5.50 mysqlpp::ReadTimeoutOption Class Reference

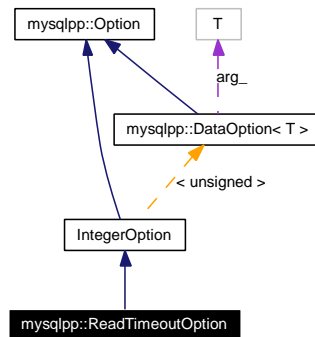
**Set**(p. 173) timeout for IPC data reads.

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

Inheritance diagram for mysqlpp::ReadTimeoutOption:



Collaboration diagram for mysqlpp::ReadTimeoutOption:



### 5.50.1 Detailed Description

**Set**(p. 173) timeout for IPC data reads.

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

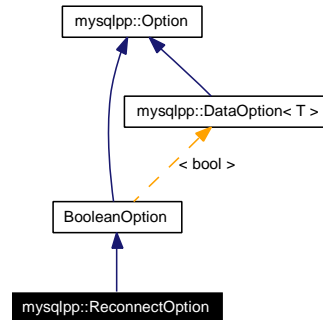
- `options.h`

## 5.51 mysqlpp::ReconnectOption Class Reference

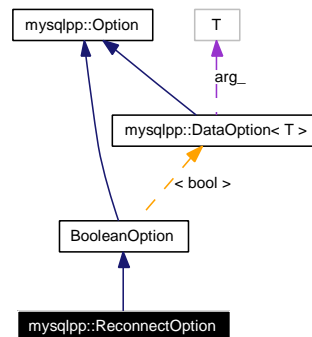
Enable automatic reconnection to server.

```
#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`

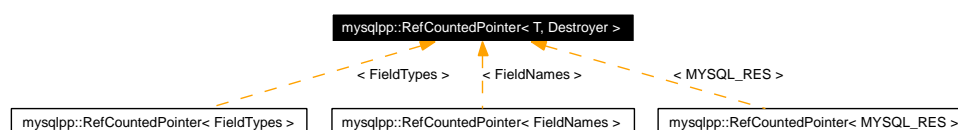


## 5.52 mysqlpp::RefCountedPointer< T, Destroyer > Class Template Reference

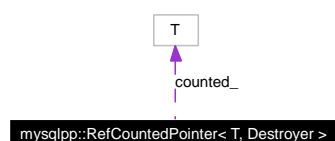
Creates an object that acts as a reference-counted pointer to another object.

```
#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. 173) (or reset) the pointer to the counted object.*
- **ThisType & operator=** (const **ThisType** &rhs)  
*Copy an existing refcounted pointer.*
- **T \* operator**  $\rightarrow$  () 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.

## 5.52 mysqlpp::RefCountedPointer< T, Destroyer > Class Template Referenced 47

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 = RefCountedPointer-  
Destroyer<T>> mysqlpp::RefCountedPointer< T, Destroyer  
>::RefCountedPointer () [inline]`

Default constructor.

An object constructed this way is useless until you vivify it with **operator =()**(p. 149) or **assign()**(p. 148).

**5.52.2.2** `template<class T, class Destroyer = RefCountedPointer-  
Destroyer<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. 149) or **assign()**(p. 148).

**5.52.2.3** `template<class T, class Destroyer = RefCountedPointer-Destroyer<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 = RefCountedPointer-Destroyer<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.

**5.52.3.2** `template<class T, class Destroyer = RefCountedPointer-Destroyer<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.

**5.52.3.3** `template<class T, class Destroyer = RefCountedPointer-Destroyer<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*()`(p. 149)

## 5.52 mysqlpp::RefCountedPointer< T, Destroyer > Class Template Referenced 49

**5.52.3.4** `template<class T, class Destroyer = RefCountedPointer-  
Destroyer<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. 146) instead. It's used internally by the compiler to implement operators bool, ==, and !=

**WARNING:** This makes it possible to say

```
///
```

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.

**5.52.3.5** `template<class T, class Destroyer = RefCountedPointer-  
Destroyer<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. 148). The choice between the two is just a matter of syntactic preference.

**5.52.3.6** `template<class T, class Destroyer = RefCountedPointer-  
Destroyer<T>> ThisType& mysqlpp::RefCountedPointer< T,  
Destroyer >::operator= (T * c) [inline]`

**Set**(p. 173) (or reset) the pointer to the counted object.

This is essentially the same thing as **assign(T\*)(p. 148)**. The choice between the two is just a matter of syntactic preference.

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. 145). You won't use this directly, you'll pass a functor of your own devising for the second parameter to the **RefCountedPointer**(p. 145) 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:

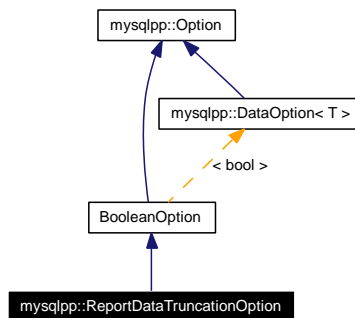
- **result.h**

## 5.55 mysqlpp::ReportDataTruncationOption Class Reference

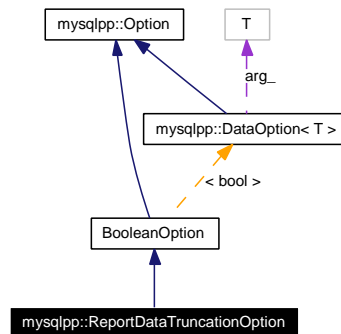
**Set**(p. 173) reporting of data truncation errors.

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

Inheritance diagram for mysqlpp::ReportDataTruncationOption:



Collaboration diagram for mysqlpp::ReportDataTruncationOption:



### 5.55.1 Detailed Description

**Set**(p. 173) reporting of data truncation errors.

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

- `options.h`

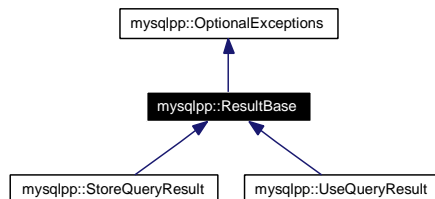


## 5.56 mysqlpp::ResultBase Class Reference

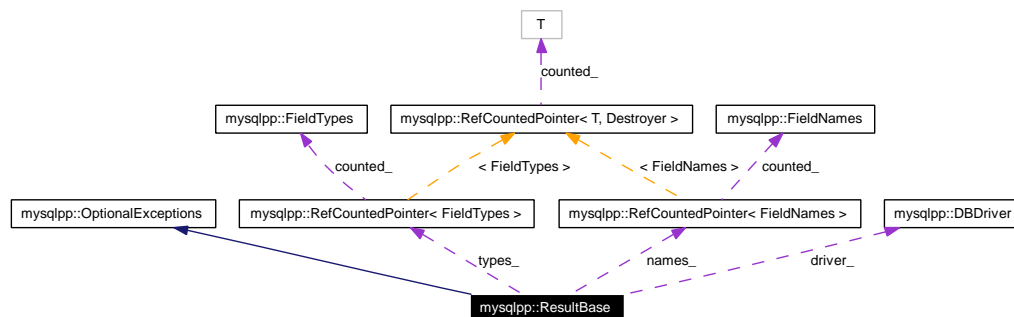
Base class for **StoreQueryResult**(p. 198) and **UseQueryResult**(p. 234).

```
#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. 81) 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. 153).*
- **ResultBase & copy (const ResultBase &other)**  
*Copy another ResultBase(p. 153) 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. 153).*

### 5.56.1 Detailed Description

Base class for **StoreQueryResult**(p. 198) and **UseQueryResult**(p. 234).

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 &) const`

Get the index of the named field.

This is the inverse of **field\_name()**(p. 154).

### 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. 153).

It's mutable because it's just internal housekeeping: it's changed by **fetch\_field(void)**(p. 153), 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.

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

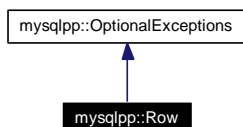
- **result.h**
- result.cpp

## 5.57 mysqlpp::Row Class Reference

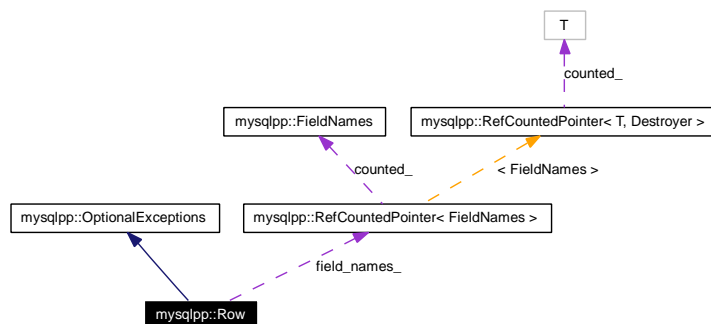
Manages rows from a result set.

```
#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. 200). It adds stuff for populating the vector. As for why it's `const`, what would it mean to modify a **Row**(p. 157)? 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 interface independent of that of `vector`.

### 5.57.2 Member Typedef Documentation

#### 5.57.2.1 `typedef const_iterator mysqlpp::Row::iterator`

iterator type

Note that this is just an alias for the `const` iterator. **Row**(p. 157) 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.

#### 5.57.2.2 `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.

#### 5.57.2.3 `typedef const_reference mysqlpp::Row::reference`

reference type

**See also:**

**iterator**(p. 162) for justification for this **const\_reference**(p. 157) alias

#### 5.57.2.4 `typedef const_reverse_iterator mysqlpp::Row::reverse_iterator`

mutable reverse iterator type

**See also:**

**iterator**(p. 162) for justification for this **const\_reverse\_iterator**(p. 157) 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

### 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.

#### 5.57.4.2 template<class Manip> equal\_list\_ba< FieldNames, Row, Manip > mysqlpp::Row::equal\_list (const char \* *d*, const char \* *e*, Manip *m*) const

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:

```
///
```

Notice how the single quote was 'escaped' in the SQL way to avoid a syntax error.

**5.57.4.3** `equal_list_b< 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.

**5.57.4.4** `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.

**5.57.4.5** `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.

**5.57.4.6** `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`

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.

**5.57.4.7** `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. 81) names will be quoted and escaped when inserted into a C++ stream, and a comma will be placed between them as a delimiter.

**5.57.4.8** `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. 81) names will be quoted and escaped when inserted into a C++ stream.

**5.57.4.9** `template<class Manip> value_list_b< FieldNames, Manip >  
mysqlpp::Row::field_list (const char * d, Manip m, 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  
*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

**5.57.4.10** `template<class Manip> value_list_ba< FieldNames, Manip >  
mysqlpp::Row::field_list (const char * d, Manip m) 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  
*m* manipulator to use before each item when the list is inserted into a C++ stream

#### 5.57.4.11 `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.

#### 5.57.4.12 `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. 157) in bool context, which lets you do things like tell when you've run off the end of a "use" query's result set:

```
///
```

#### 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. 163) 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 `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.

**5.57.4.15** `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.

**5.57.4.16** `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.

**5.57.4.17** `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.

**5.57.4.18** `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.

**5.57.4.19** `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** `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.

**5.57.4.21** `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.



**5.57.4.22** `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

**5.57.4.23** `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

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

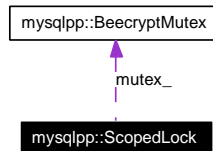
- row.h
- row.cpp

## 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:

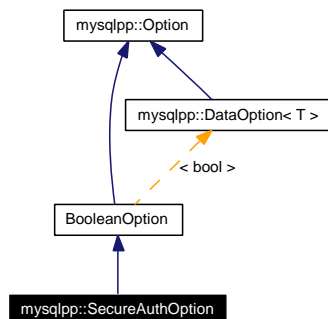
- **beemutex.h**

## 5.59 mysqlpp::SecureAuthOption Class Reference

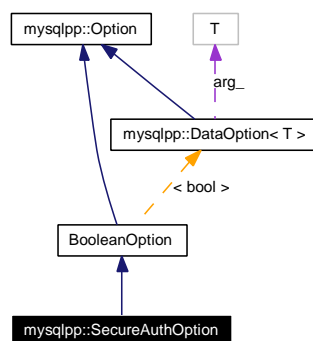
Enforce use of secure authentication, refusing connection if not available.

```
#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:

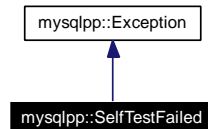
- **options.h**

## 5.60 mysqlpp::SelfTestFailed Class Reference

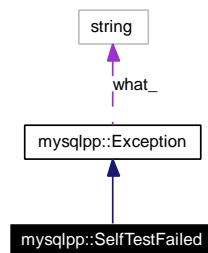
Used within MySQL++'s test harness only.

```
#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:

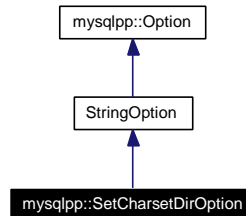
- **myset.h**

## 5.62 mysqlpp::SetCharsetDirOption Class Reference

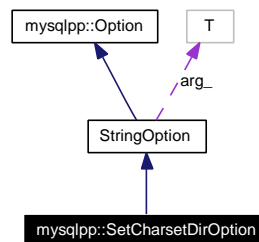
Give path to charset definition files.

```
#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:

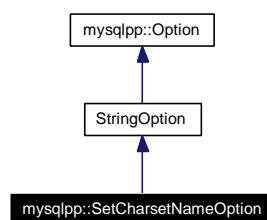
- **options.h**

## 5.63 mysqlpp::SetCharsetNameOption Class Reference

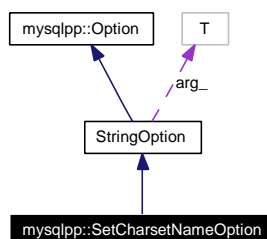
Give name of default charset.

```
#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:

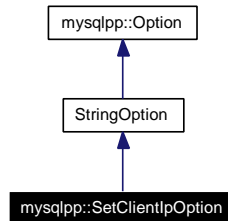
- `options.h`

## 5.64 mysqlpp::SetClientIpOption Class Reference

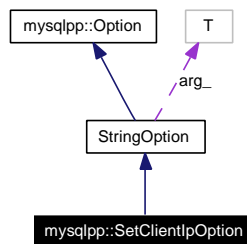
Fake client IP address when connecting to embedded server.

```
#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**

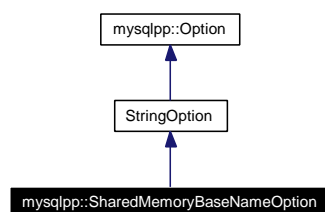


## 5.65 mysqlpp::SharedMemoryBaseNameOption Class Reference

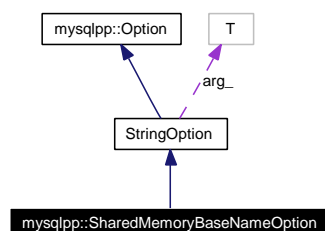
**Set**(p. 173) name of shmем segment for IPC.

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

Inheritance diagram for mysqlpp::SharedMemoryBaseNameOption:



Collaboration diagram for mysqlpp::SharedMemoryBaseNameOption:



### 5.65.1 Detailed Description

**Set**(p. 173) name of shmем segment for IPC.

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

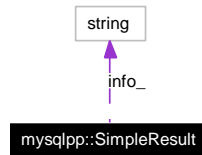
- **options.h**

## 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. 125) or **Query::errnum()**(p. 125) to find out what exactly happened.

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

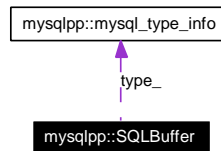
- **result.h**

## 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 `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".

#### 5.67.3.2 `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.

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

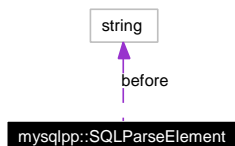
- `sql_buffer.h`
- `sql_buffer.cpp`

## 5.68 mysqlpp::SQLParseElement Struct Reference

Used within **Query**(p. 119) 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. 119) 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. 183) 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. 119) 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**

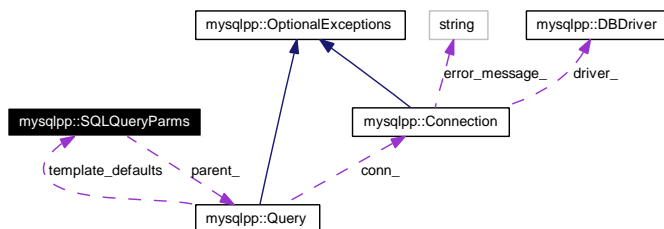


## 5.69 mysqlpp::SQLQueryParms Class Reference

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. 126).*
- size\_t **escape\_string** (char \*escaped, const char \*original, size\_t length) const  
*Indirect access to **Query::escape\_string**(p. 126).*

- **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. 173) 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. 126).

**See also:**

`escape_string(std::string*, const char*, size_t)`

`Query::escape_string(const char*, const char*, size_t)`

**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. 173) 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:

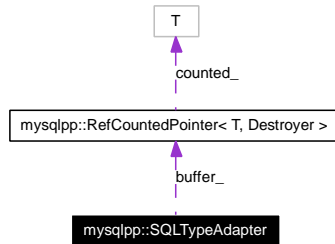
- `qparms.h`
- `qparms.cpp`

## 5.70 mysqlpp::SQLTypeAdapter Class Reference

Converts many different data types to strings suitable for use in SQL queries.

```
#include <stadapter.h>
```

Collaboration diagram for mysqlpp::SQLTypeAdapter:



### 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. 191)*
- 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. 119) 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. 188), 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. 188). The conversion to **SQLTypeAdapter**(p. 188) 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. 188) 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. 201) or **String::conv**(p. 201) 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. 220) instead. It avoids the confusion in C++ between integer and character. See the documentation for **tiny\_int.h**(p. 302) for details.



### 5.70.3 Member Function Documentation

#### 5.70.3.1 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*

#### 5.70.3.2 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.

#### 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. 193) overloads taking a C or C++ string instead.

**5.70.3.4 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

**5.70.3.5 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\*).

**5.70.3.6 int mysqlpp::SQLTypeAdapter::compare (const char \* *other*) const**

Compare the internal buffer to the given string.

Works just like string::compare(const char\*).

**5.70.3.7 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&).

**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&).

**5.70.3.9 int mysqlpp::SQLTypeAdapter::compare (const SQLTypeAdapter & *other*) const**

Compare the internal buffer to the given string.

Works just like string::compare(const std::string&).

**5.70.3.10 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.

**5.70.3.11 SQLTypeAdapter & mysqlpp::SQLTypeAdapter::operator= (const null\_type & n)**

Replace contents of object with a SQL null.

**See also:**

`assign(const null_type&)`(p. 193) for details

**5.70.3.12 SQLTypeAdapter & mysqlpp::SQLTypeAdapter::operator= (const SQLTypeAdapter & rhs)**

Standard assignment operator.

**See also:**

`assign(const SQLTypeAdapter&)`(p. 193) for details

**5.70.3.13 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.

**5.70.3.14 int mysqlpp::SQLTypeAdapter::type\_id () const**

Returns the type ID of the buffer's data.

Values from `type_info.h`(p. 305). 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.

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

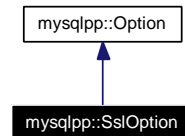
- `stadapter.h`
- `stadapter.cpp`

## 5.71 mysqlpp::SslOption Class Reference

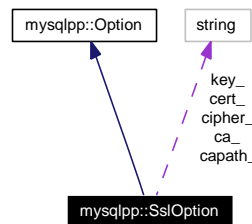
Specialized option for handling SSL parameters.

```
#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.

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:

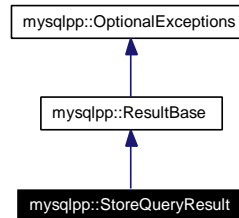
- **options.h**

## 5.72 mysqlpp::StoreQueryResult Class Reference

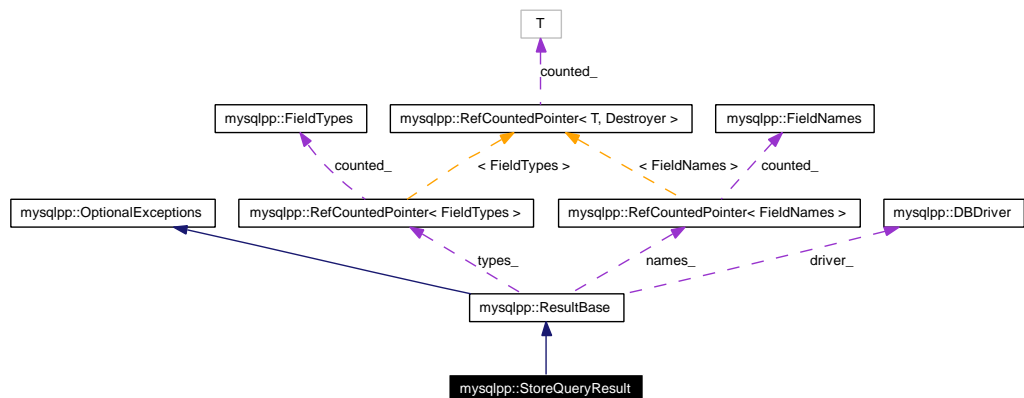
**StoreQueryResult**(p. 198) set type for "store" queries.

```
#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. 198) 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. 198) 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. 198) 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. 157) objects in memory so you get random-access semantics. MySQL++ also supports **UseQueryResult**(p. 234) 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. 125) or **Query::errnum**(p. 125) to find out what exactly happened.

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

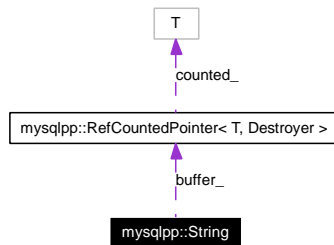
- **result.h**
- **result.cpp**

## 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>
```

Collaboration diagram for mysqlpp::String:



### 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. 201) 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. 173) 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. 200).*
- **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. 50).*
- **operator DateTime () const**  
*Converts this object's string data to a `mysqlpp::DateTime`(p. 54).*
- **operator Time () const**  
*Converts this object's string data to a `mysqlpp::Time`(p. 216).*
- **template<class T, class B> operator Null () const**  
*Converts the `String`(p. 200) 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. 200) 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. 297), 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. 200) 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. 200) 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, 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. 200) temporary to get around the fact that **operator=**(p. 203) 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. 200) temporary to get around the fact that **operator=**(p. 203) can only take one parameter.

**5.73.3.3** `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. 200) temporary to get around the fact that **operator=**(p. 203) 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. 211), this function throws an `std::out_of_range` exception if the index isn't within range.

**5.73.3.5 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

**5.73.3.6 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\*)

**5.73.3.7 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\*)



**5.73.3.8 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\*)

**5.73.3.9 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\*)

**5.73.3.10 template<class T, class B> Null<T, B> mysqlpp::String::conv (Null< T, B >) const [inline]**

Overload of **conv()**(p. 201) for types wrapped with Null<>.

If the **String**(p. 200) 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.

**5.73.3.11 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.

**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. 211); its size is always equal to the amount of data currently stored.

**5.73.3.13** `template<class T, class B> mysqlpp::String::operator Null () const`  
`[inline]`

Converts the **String**(p. 200) to a nullable data type.

This is just an implicit version of `conv(Null<T, B>)`

**5.73.3.14** `bool mysqlpp::String::operator!= (const mysqlpp::null_type &) const`  
`[inline]`

Inequality comparison operator.

For checking object against MySQL++'s global `null` constant

**5.73.3.15** `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. 209) overload for.

**5.73.3.16** `String& mysqlpp::String::operator= (const String & other)`  
`[inline]`

Assignment operator, from other **String**(p. 200).

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.

**5.73.3.17** `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.

**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]`

Equality comparison operator.

For comparing this object to any of the data types we have a **compare()**(p. 209) overload for.

**5.73.3.20** `char mysqlpp::String::operator[] (size_type pos) const`

Return a character within the string.

Unlike **at()**(p. 207), this access method doesn't check the index for sanity.

**5.73.3.21** `size_type mysqlpp::String::size () const [inline]`

Return number of bytes in string.

See commentary for **length()**(p. 209) about the difference between bytes and characters.

**5.73.3.22** `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. 202) method.) This copies a pointer to a buffer instead of copying the buffer's contents.

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

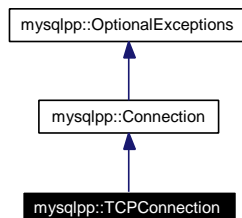
- **mystring.h**
- **mystring.cpp**

## 5.74 mysqlpp::TCPConnection Class Reference

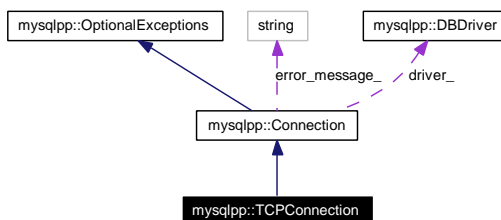
Specialization of **Connection**(p. 31) for TCP/IP.

```
#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. 31) for TCP/IP.

This class just simplifies the connection creation interface of **Connection**(p. 31). 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. 31). 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.

#### 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

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

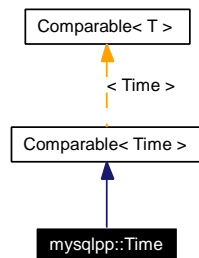
- **tcp\_connection.h**
- tcp\_connection.cpp

## 5.75 mysqlpp::Time Class Reference

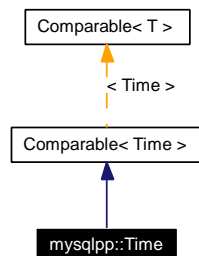
C++ form of SQL's TIME type.

```
#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. 216).*
- **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. 200) 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. 200)...

See also:

`Time(const char*)(p. 218)`

#### 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. 54) 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` if they are equal, and `> 0` if this time is after the other.

### 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>
```

### 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

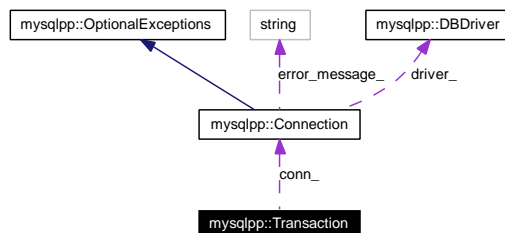


## 5.78 mysqlpp::Transaction Class Reference

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.

**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.

**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.

**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.

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

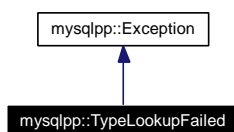
- **transaction.h**
- **transaction.cpp**

## 5.79 mysqlpp::TypeLookupFailed Class Reference

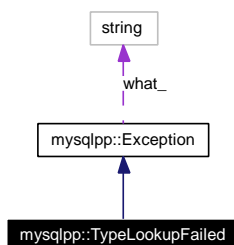
Thrown from the C++ to SQL data type conversion routine when it can't figure out how to map the type.

```
#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:

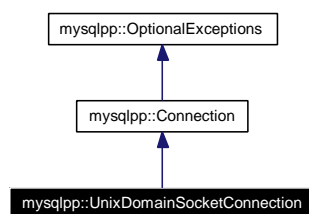
- `exceptions.h`

## 5.80 mysqlpp::UnixDomainSocketConnection Class Reference

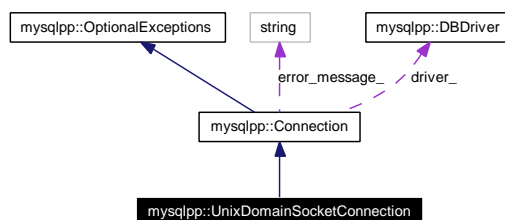
Specialization of **Connection**(p. 31) for Unix domain sockets.

```
#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. 31) for Unix domain sockets.

This class just simplifies the connection creation interface of **Connection**(p. 31). 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. 31). This is a feature, not a bug. :)

- 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.

**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

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

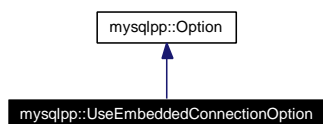
- `uds_connection.h`
- `uds_connection.cpp`

## 5.81 mysqlpp::UseEmbeddedConnectionOption Class Reference

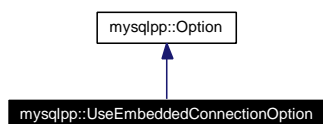
Connect to embedded server in preference to remote server.

```
#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**

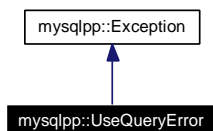


## 5.82 mysqlpp::UseQueryError Class Reference

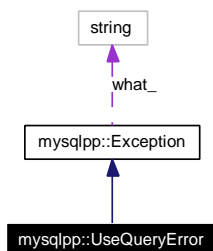
**Exception**(p. 79) thrown when something goes wrong in processing a "use" query.

```
#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. 79) thrown when something goes wrong in processing a "use" query.

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

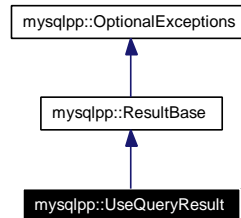
- **exceptions.h**

## 5.83 mysqlpp::UseQueryResult Class Reference

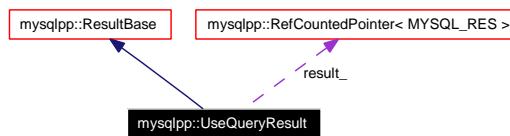
**StoreQueryResult**(p. 198) set type for "use" queries.

```
#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. 234) object.*
- **~UseQueryResult ()**  
*Destroy object.*
- **UseQueryResult & operator= (const UseQueryResult &rhs)**  
*Copy another UseQueryResult(p. 234) 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. 198) 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. 67). It does a lot of error checking before returning the **Row**(p. 157) object containing the row data.

See also:

**fetch\_raw\_row**(p. 235)

#### 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. 153).

#### 5.83.2.3 `mysqlpp::UseQueryResult::operator MYSQL_RES * () const` [inline]

Return the pointer to the underlying MySQL C API result set object.

///

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

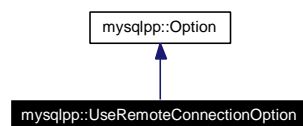
- `result.h`
- `result.cpp`

## 5.84 mysqlpp::UseRemoteConnectionOption Class Reference

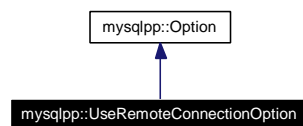
Connect to remote server in preference to embedded server.

```
#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:

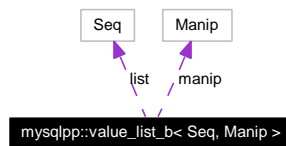
- `options.h`

## 5.85 mysqlpp::value\_list\_b< Seq, Manip > Struct Template Reference

Same as **value\_list\_ba**(p. 240), 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. 240), 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:

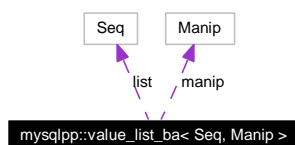
- **vallist.h**

## 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 SSQSL mechanism calls those functions when building SQL queries; you can call them yourself to do similar work. The "Harnessing SSQSL Internals" section of the user manual has some examples of this.

**See also:**

`value_list_b`(p. 238)

## 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:

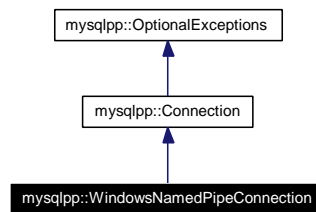
- `vallist.h`

## 5.87 mysqlpp::WindowsNamedPipeConnection Class Reference

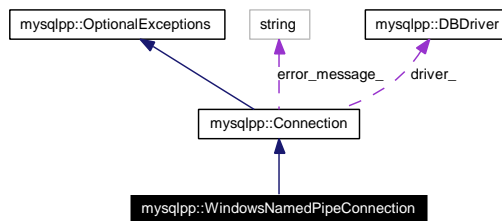
Specialization of **Connection**(p. 31) for Windows named pipes.

```
#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. 31) for Windows named pipes.

This class just simplifies the connection creation interface of **Connection**(p. 31). 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.

#### 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

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

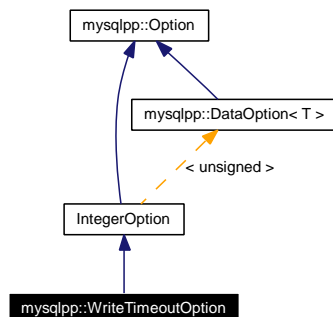
- `wnp_connection.h`
- `wnp_connection.cpp`

## 5.88 mysqlpp::WriteTimeoutOption Class Reference

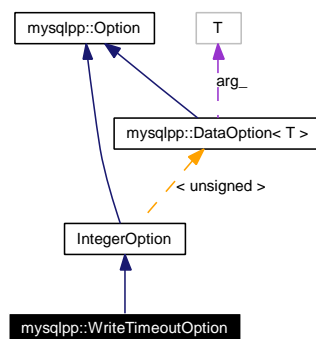
**Set**(p. 173) timeout for IPC data reads.

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

Inheritance diagram for mysqlpp::WriteTimeoutOption:



Collaboration diagram for mysqlpp::WriteTimeoutOption:



### 5.88.1 Detailed Description

**Set**(p. 173) timeout for IPC data reads.

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

- `options.h`



## Chapter 6

# MySQL++ 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

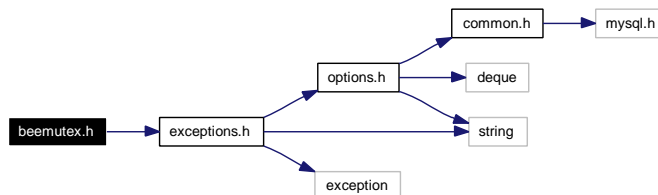
Defines a template for setting a flag within a given variable scope, and resetting it when exiting that scope.

## 6.2 beemutex.h File Reference

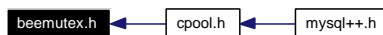
MUTually EXclusive lock class.

```
#include "exceptions.h"
```

Include dependency graph for beemutex.h:



This graph shows which files directly or indirectly include this file:



### 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

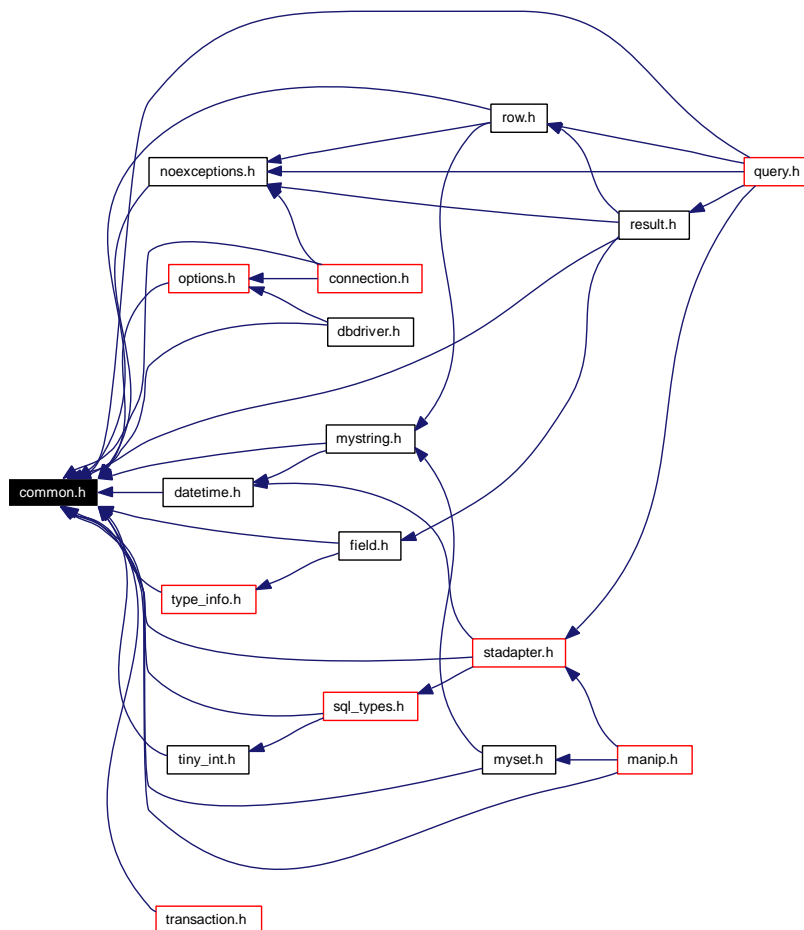
This file includes top-level definitions for use both internal to the library, and outside it. Contrast `mysql++.h`.

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

Include dependency graph for `common.h`:



This graph shows which files directly or indirectly include this file:



### 6.3.1 Detailed Description

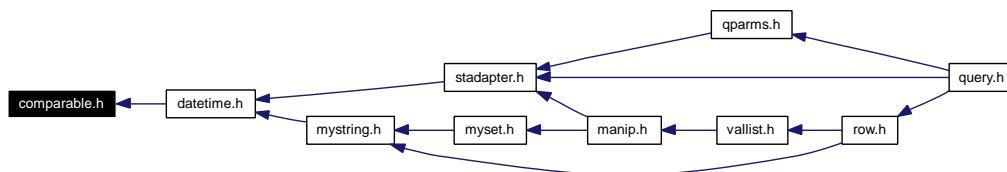
This file includes top-level definitions for use both internal to the library, and outside it. Contrast mysql++.h.

This file mostly takes care of platform differences.

## 6.4 comparable.h File Reference

Declares the `Comparable<T>` mixin.

This graph shows which files directly or indirectly include this file:



### Classes

- class **Comparable**< T >

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

### 6.4.1 Detailed Description

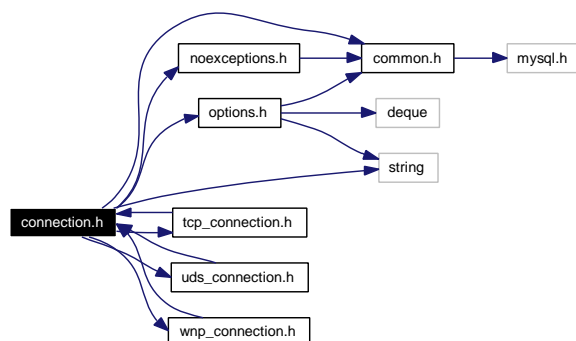
Declares the `Comparable<T>` mixin.

## 6.5 connection.h File Reference

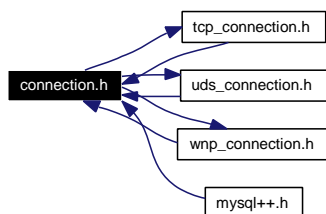
Declares the Connection class.

```
#include "common.h"
#include "noexceptions.h"
#include "options.h"
#include <string>
#include "tcp_connection.h"
#include "uds_connection.h"
#include "wnp_connection.h"
```

Include dependency graph for connection.h:



This graph shows which files directly or indirectly include this file:



## Namespaces

- namespace **mysqlpp**

## Classes

- class **mysqlpp::Connection**

*Manages the connection to the database server.*

### 6.5.1 Detailed Description

Declares the Connection class.

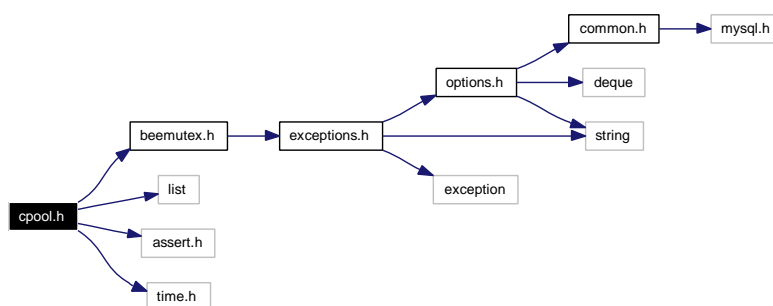
Every program using MySQL++ must create a Connection object, which manages information about the connection to the database server, and performs connection-related operations once the connection is up. Subordinate classes, such as Query and Row take their defaults as to whether exceptions are thrown 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>
```

Include dependency graph for cpool.h:



This graph shows which files directly or indirectly include this file:



### Namespaces

- namespace **mysqlpp**

### Classes

- class **mysqlpp::ConnectionPool**  
*Manages a pool of connections for programs that need more than one **Connection**(p. 31) 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"
```

Include dependency graph for custom.h:



### 6.7.1 Detailed Description

Backwards-compatibility header; loads ssqls.h.

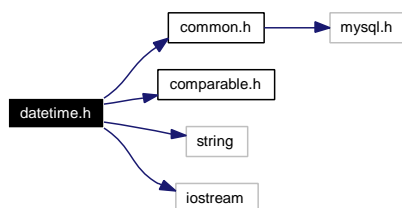


## 6.8 datetime.h File Reference

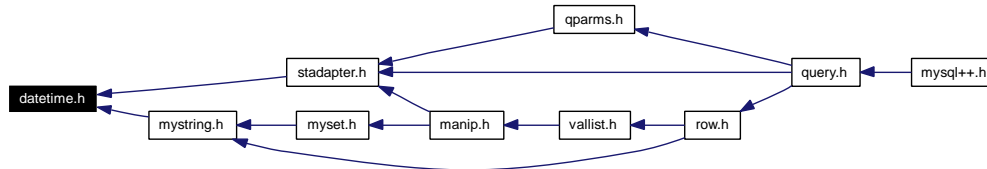
Declares classes to add SQL-compatible date and time types to C++'s type system.

```
#include "common.h"
#include "comparable.h"
#include <string>
#include <iostream>
```

Include dependency graph for datetime.h:



This graph shows which files directly or indirectly include this file:



## 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. 54) object into a C++ stream in a SQL-compatible format.*
- `std::ostream & mysqlpp::operator<< (std::ostream &os, const Date &d)`  
*Inserts a **Date**(p. 50) object into a C++ stream.*
- `std::ostream & mysqlpp::operator<< (std::ostream &os, const Time &t)`  
*Inserts a **Time**(p. 216) object into a C++ stream in a SQL-compatible format.*

### 6.8.1 Detailed Description

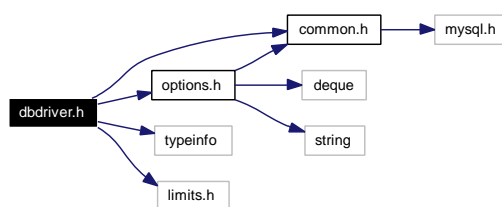
Declares classes to add SQL-compatible date and time types to C++'s type system.

## 6.9 dbdriver.h File Reference

Declares the DBDriver class.

```
#include "common.h"
#include "options.h"
#include <typeinfo>
#include <limits.h>
```

Include dependency graph for dbdriver.h:



### Namespaces

- namespace **mysqlpp**

### Classes

- class **mysqlpp::DBDriver**

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

#### 6.9.1 Detailed Description

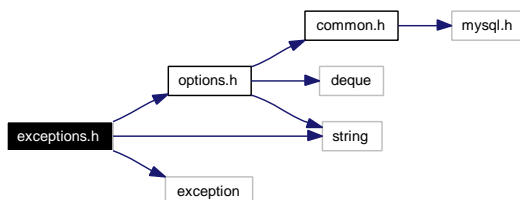
Declares the DBDriver class.

## 6.10 exceptions.h File Reference

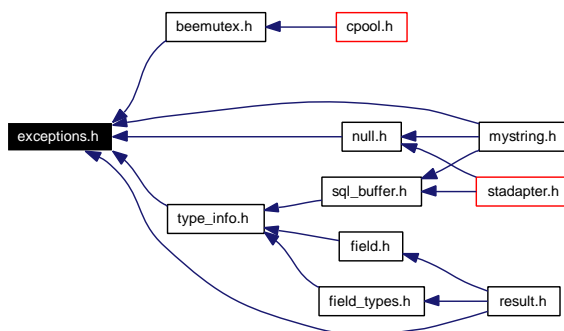
Declares the MySQL++-specific exception classes.

```
#include "options.h"
#include <exception>
#include <string>
```

Include dependency graph for exceptions.h:



This graph shows which files directly or indirectly include this file:



### Namespaces

- namespace **mysqlpp**

### Classes

- class **mysqlpp::Exception**  
*Base class for all MySQL++ custom exceptions.*
- class **mysqlpp::BadConversion**

**Exception**(p. 79) thrown when a bad type conversion is attempted.

- class **mysqlpp::BadFieldName**

**Exception**(p. 79) thrown when a requested named field doesn't exist.

- class **mysqlpp::BadOption**

**Exception**(p. 79) thrown when you pass an unrecognized option to **Connection::set\_option()**(p. 39).

- class **mysqlpp::BadParamCount**

**Exception**(p. 79) thrown when not enough query parameters are provided.

- class **mysqlpp::UseQueryError**

**Exception**(p. 79) thrown when something goes wrong in processing a "use" query.

- class **mysqlpp::BadQuery**

**Exception**(p. 79) thrown when the database server encounters a problem while processing your query.

- class **mysqlpp::ConnectionFailed**

**Exception**(p. 79) thrown when there is a problem related to the database server connection.

- class **mysqlpp::DBSelectionFailed**

**Exception**(p. 79) thrown when the program tries to select a new database and the database server refuses for some reason.

- class **mysqlpp::MutexFailed**

**Exception**(p. 79) thrown when a **BeecryptMutex**(p. 26) object fails.

- class **mysqlpp::ObjectNotInitialized**

**Exception**(p. 79) 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.

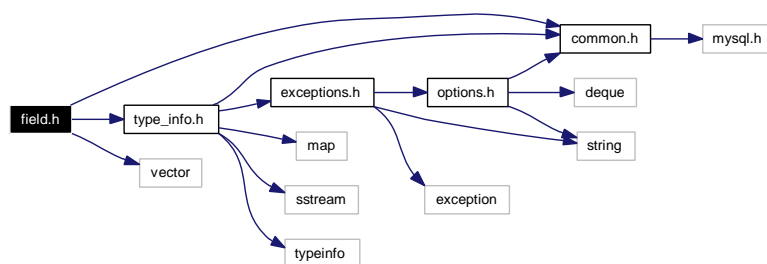
When exceptions are enabled for a given `mysqlpp::OptionalExceptions`(p. 117) derivative, any of these exceptions can be thrown on error.

## 6.11 field.h File Reference

Declares the Field and Fields classes.

```
#include "common.h"
#include "type_info.h"
#include <vector>
```

Include dependency graph for field.h:



This graph shows which files directly or indirectly include this file:



### 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.

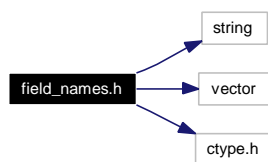


## 6.12 field\_names.h File Reference

Declares a class to hold a list of field names.

```
#include <string>
#include <vector>
#include <ctype.h>
```

Include dependency graph for field\_names.h:



This graph shows which files directly or indirectly include this file:



### 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.

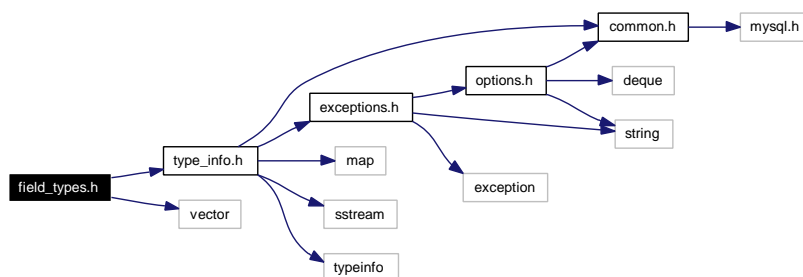
## 6.13 field\_types.h File Reference

Declares a class to hold a list of SQL field type info.

```
#include "type_info.h"
```

```
#include <vector>
```

Include dependency graph for field\_types.h:



This graph shows which files directly or indirectly include this file:



### Namespaces

- namespace **mysqlpp**

### Classes

- class **mysqlpp::FieldTypes**

*A vector of SQL field types.*

#### 6.13.1 Detailed Description

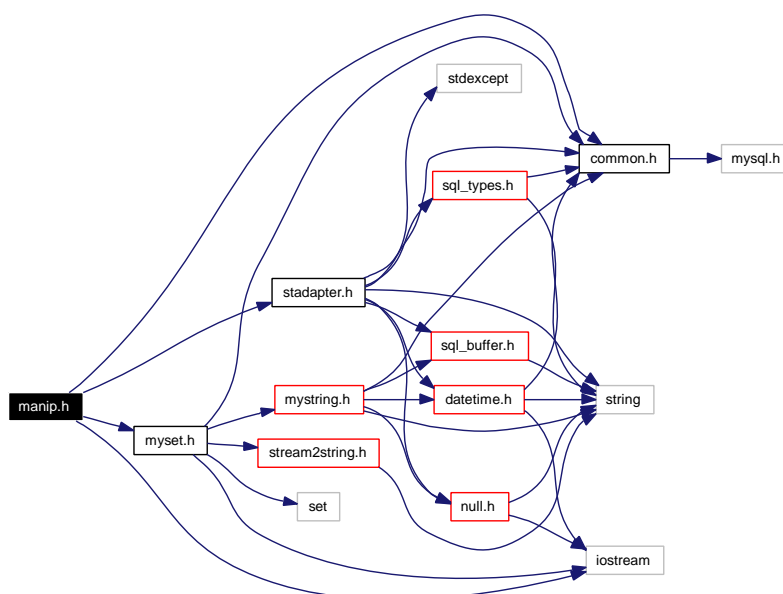
Declares a class to hold a list of SQL field type info.

## 6.14 manip.h File Reference

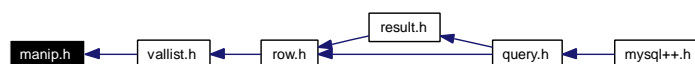
Declares the Query stream manipulators and operators.

```
#include "common.h"
#include "myset.h"
#include "stadapter.h"
#include <iostream>
```

Include dependency graph for manip.h:



This graph shows which files directly or indirectly include this file:



## 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. 188) 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. 188) 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.

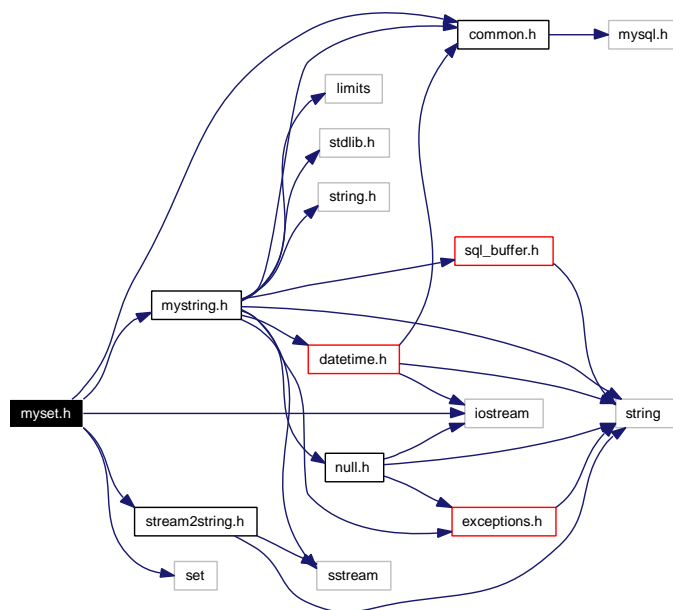
**test/test\_manip.cpp** exercises the mechanisms defined here.

## 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>
```

Include dependency graph for myset.h:



This graph shows which files directly or indirectly include this file:



## 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. 173) object into a C++ stream.*

### 6.15.1 Detailed Description

Declares templates for generating custom containers used elsewhere in the library.

The main MySQL++ header file.

```
#include "connection.h"
#include "cpool.h"
#include "query.h"
#include "sql_types.h"
#include "transaction.h"
```

```

graph LR
    mysql++.h --> cpool.h
    mysql++.h --> query.h
    mysql++.h --> transaction.h
    cpool.h --> time.h
    cpool.h --> beemutex.h
    cpool.h --> assert.h
    query.h --> list
    query.h --> qparms.h
    query.h --> map
    query.h --> set
    query.h --> result.h
    query.h --> querydef.h
    query.h --> iomanip
    query.h --> vector
    query.h --> row.h
    query.h --> sql_types.h
    query.h --> options.h
    query.h --> tcp_connection.h
    query.h --> uds_connection.h
    query.h --> wnp_connection.h
    query.h --> deque
    query.h --> string
    query.h --> common.h
    query.h --> noexceptions.h
    transaction.h --> common.h
    connection.h --> tcp_connection.h
    connection.h --> uds_connection.h
    connection.h --> wnp_connection.h
    connection.h --> options.h
    connection.h --> string
    connection.h --> common.h
    connection.h --> noexceptions.h
    stadapter.h --> qparms.h
    stadapter.h --> string
    common.h --> noexceptions.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, 5)`

*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.256) 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, 5)`

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.

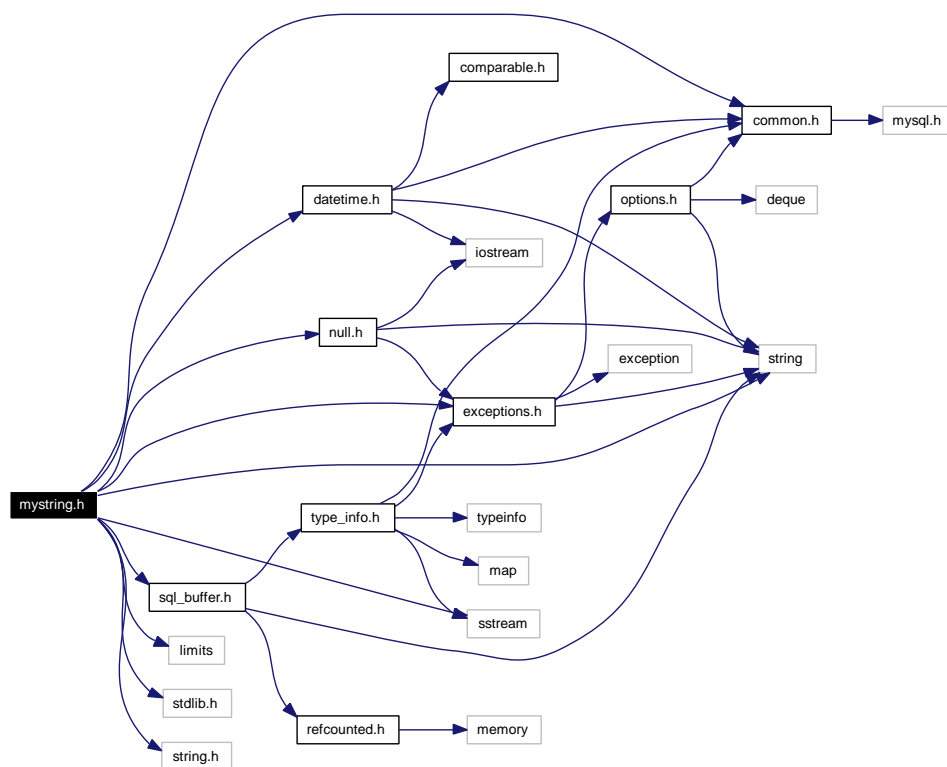
This macro takes major, minor and bugfix numbers (e.g. 1, 2, and 3) and encodes them like 0x010203.

## 6.17 mystring.h File Reference

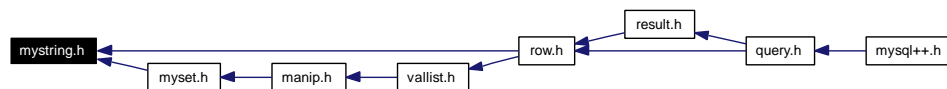
Declares String class, MySQL++'s generic std::string-like class, used for holding data received from the database server.

```
#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>
```

Include dependency graph for mystring.h:



This graph shows which files directly or indirectly include this file:



## 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. 200) objects.*

### 6.17.1 Detailed Description

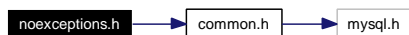
Declares String class, MySQL++'s generic `std::string`-like class, used for holding data received from the database server.

## 6.18 noexceptions.h File Reference

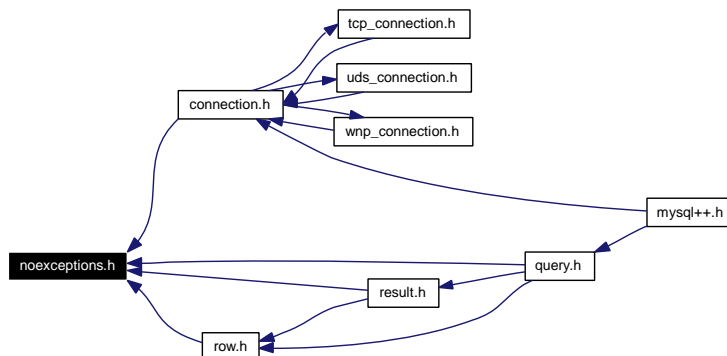
Declares interface that allows exceptions to be optional.

```
#include "common.h"
```

Include dependency graph for noexceptions.h:



This graph shows which files directly or indirectly include this file:



### 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. 117).*

#### 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

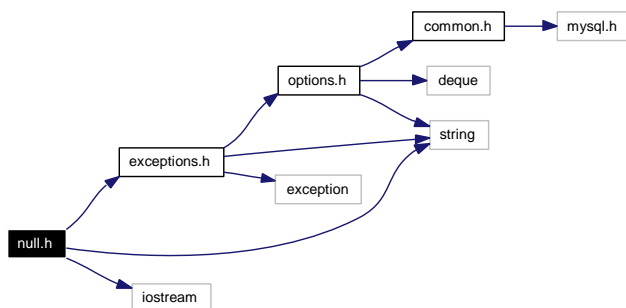
Declares classes that implement SQL "null" semantics within C++'s type system.

```
#include "exceptions.h"
```

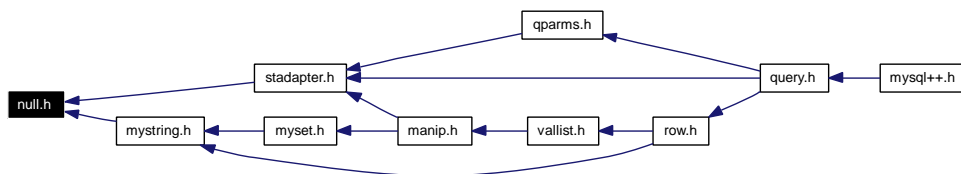
```
#include <iostream>
```

```
#include <string>
```

Include dependency graph for null.h:



This graph shows which files directly or indirectly include this file:



### 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. 110).*

- 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

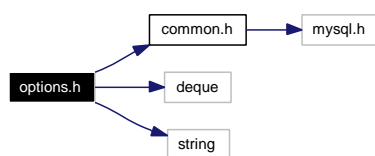
Declares the Option class hierarchy, used to implement connection options in Connection and DBDriver classes.

```
#include "common.h"
```

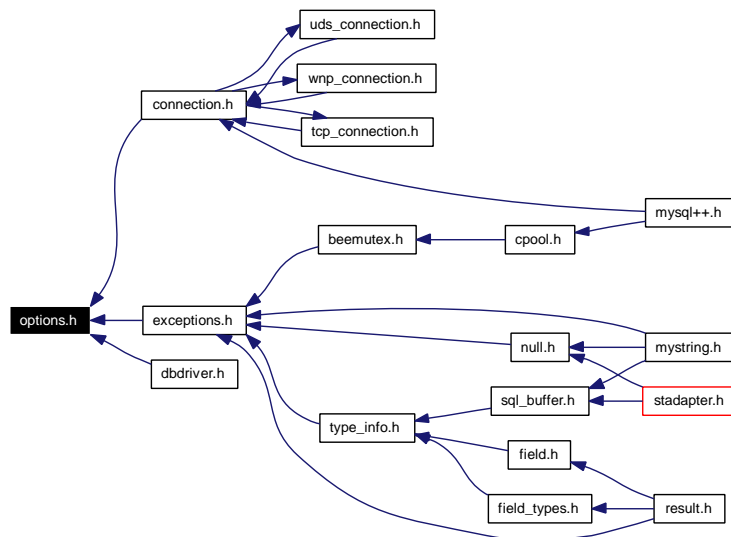
```
#include <deque>
```

```
#include <string>
```

Include dependency graph for options.h:



This graph shows which files directly or indirectly include this file:



## 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. 35) default timeout.*
- class **mysqlpp::FoundRowsOption**  
*Make **Query::affected\_rows()**(p. 119) 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. 173) timeout for IPC data reads.*
- class **mysqlpp::ReconnectOption**  
*Enable automatic reconnection to server.*
- class **mysqlpp::ReportDataTruncationOption**  
*Set(p. 173) 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. 173) 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. 173) timeout for IPC data reads.*

## Typedefs

- typedef DataOption< unsigned > **mysqlpp::IntegerOption**  
*Option(p. 115) w/ int argument.*
- typedef DataOption< bool > **mysqlpp::BooleanOption**  
*Option(p. 115) w/ bool argument.*
- typedef DataOption< std::string > **mysqlpp::StringOption**  
*Option(p. 115) 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 proportion to the importance of options.

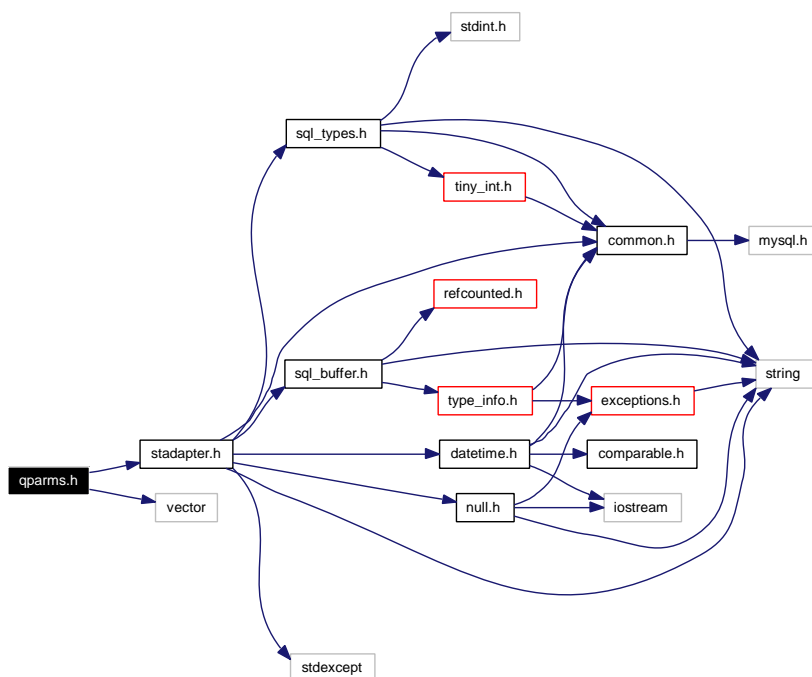
## 6.21 qparms.h File Reference

Declares the template query parameter-related stuff.

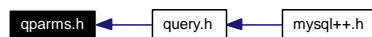
```
#include "stadapter.h"
```

```
#include <vector>
```

Include dependency graph for qparms.h:



This graph shows which files directly or indirectly include this file:



### Namespaces

- namespace **mysqlpp**

### Classes

- class **mysqlpp::SQLQueryParms**

*This class holds the parameter values for filling template queries.*

- struct **mysqlpp::SQLParseElement**

*Used within **Query**(p. 119) to hold elements for parameterized queries.*

### 6.21.1 Detailed Description

Declares the template query parameter-related stuff.

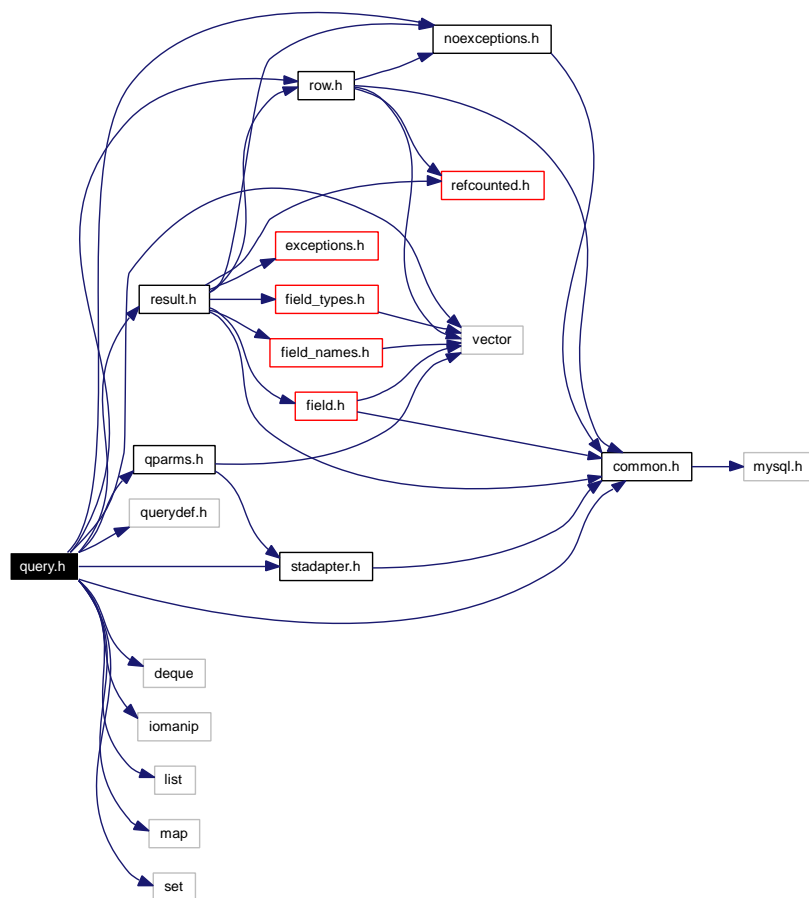
The classes defined in this file are used by class **Query** when it parses a template query: they hold information that it finds in the template, so it can assemble a SQL statement later on demand.

## 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>
```

Include dependency graph for query.h:



This graph shows which files directly or indirectly include this file:



## 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

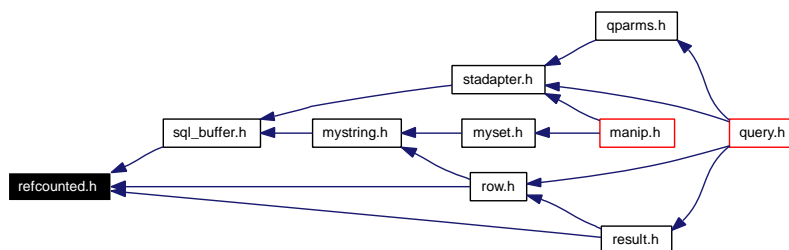
Declares the RefCountedPointer template.

```
#include <memory>
```

Include dependency graph for refcounted.h:



This graph shows which files directly or indirectly include this file:



## 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

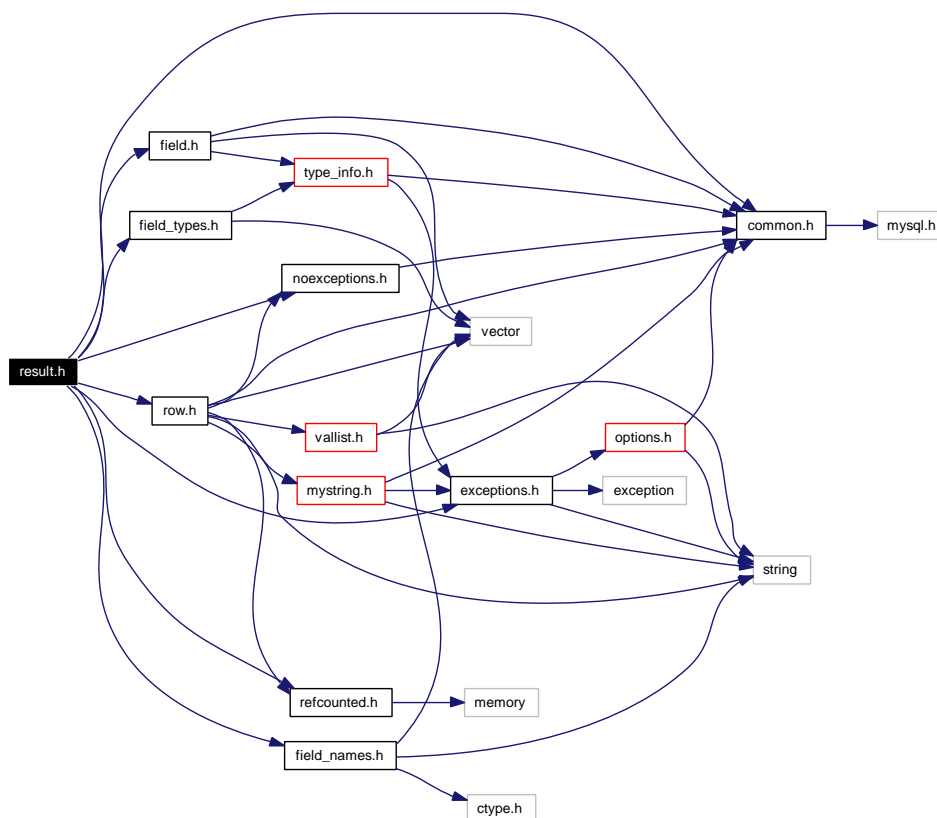
Declares the RefCountedPointer template.

## 6.24 result.h File Reference

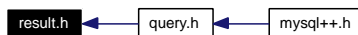
Declares classes for holding information about SQL query results.

```
#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"
```

Include dependency graph for result.h:



This graph shows which files directly or indirectly include this file:



## 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. 198) and **UseQueryResult**(p. 234).*
- class **mysqlpp::StoreQueryResult**  
***StoreQueryResult**(p. 198) 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. 198) set type for "use" queries.*

## Functions

- void **mysqlpp::swap** (StoreQueryResult &x, StoreQueryResult &y)  
*Swaps two **StoreQueryResult**(p. 198) objects.*
- void **mysqlpp::swap** (UseQueryResult &x, UseQueryResult &y)  
*Swaps two **UseQueryResult**(p. 234) objects.*

### 6.24.1 Detailed Description

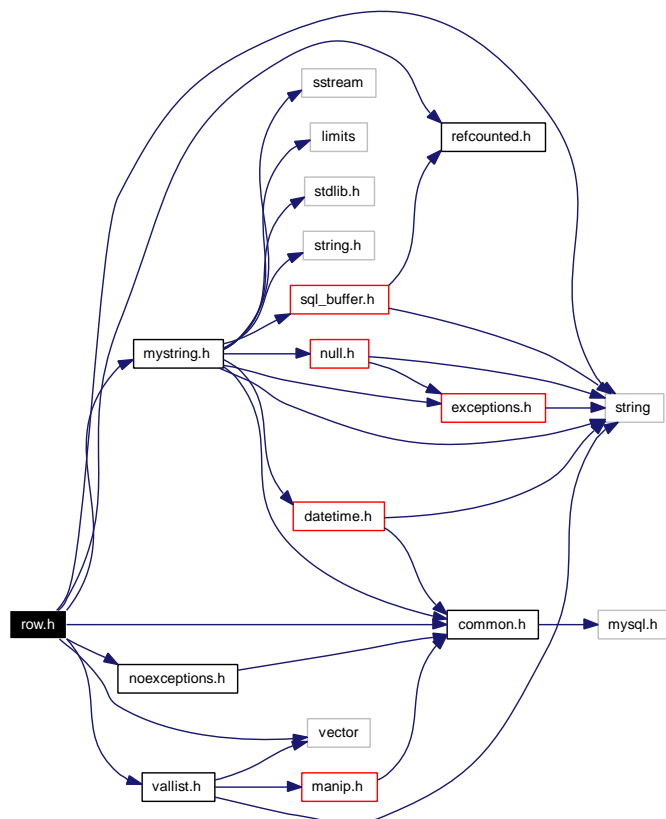
Declares classes for holding information about SQL query results.

## 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>
```

Include dependency graph for row.h:



This graph shows which files directly or indirectly include this file:



## 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

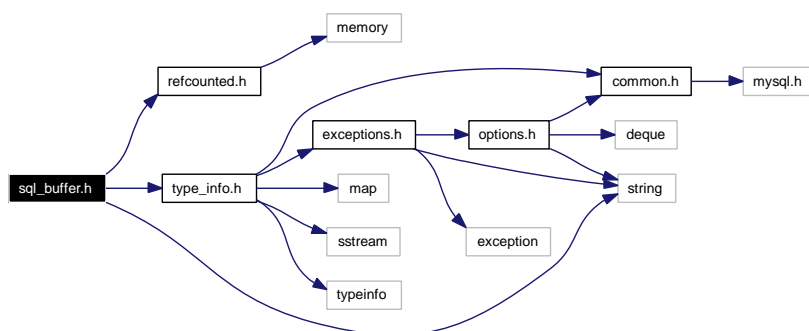
Declares the SQLBuffer class.

```
#include "refcounted.h"
```

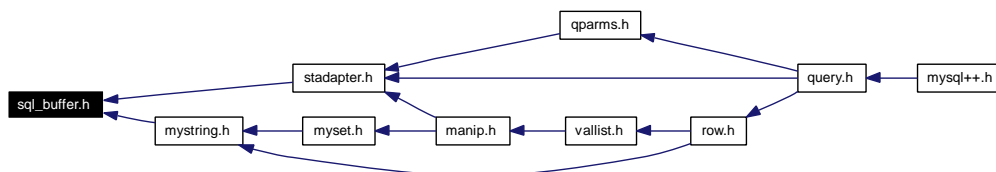
```
#include "type_info.h"
```

```
#include <string>
```

Include dependency graph for sql\_buffer.h:



This graph shows which files directly or indirectly include this file:



### 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. 180).*

### 6.26.1 Detailed Description

Declares the `SQLBuffer` class.

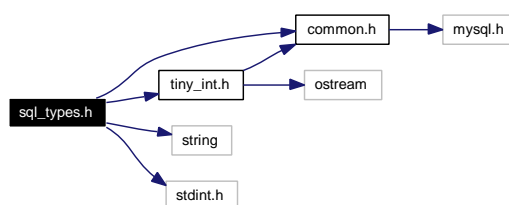


## 6.27 sql\_types.h File Reference

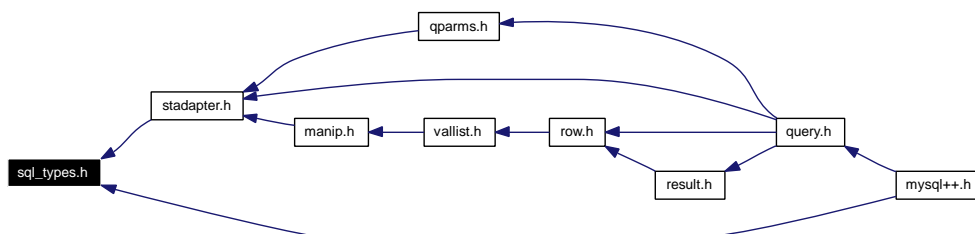
Declares the closest C++ equivalent of each MySQL column type.

```
#include "common.h"
#include "tiny_int.h"
#include <string>
#include <stdint.h>
```

Include dependency graph for sql\_types.h:



This graph shows which files directly or indirectly include this file:



## Namespaces

- namespace **mysqlpp**

### 6.27.1 Detailed Description

Declares the closest C++ equivalent of each MySQL column type.

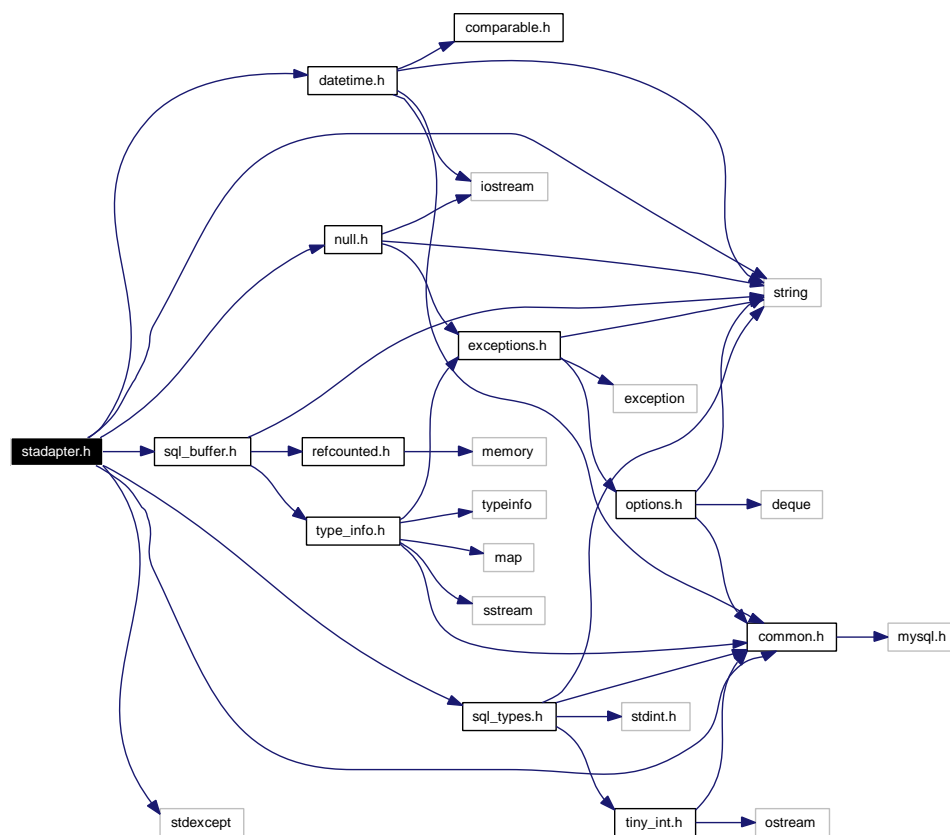
The typedefs defined here are only for the "non-NULL" variants. To get nullable versions, wrap the appropriate type in the `Null<T>` template. See **null.h**(p. 279) for more information.

## 6.28 stadapter.h File Reference

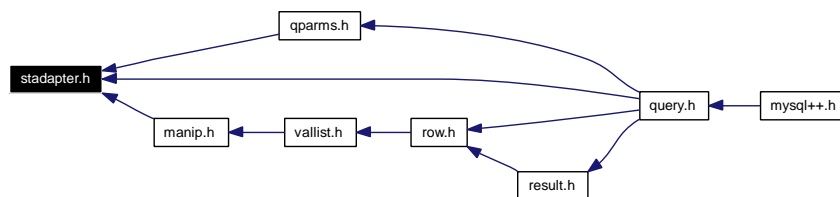
Declares the SQLTypeAdapter class.

```
#include "common.h"
#include "datetime.h"
#include "null.h"
#include "sql_buffer.h"
#include "sql_types.h"
#include <stdexcept>
#include <string>
```

Include dependency graph for stadapter.h:



This graph shows which files directly or indirectly include this file:



## 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.

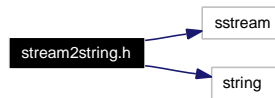
## 6.29 stream2string.h File Reference

Declares an adapter that converts something that can be inserted into a C++ stream into a `std::string` type.

```
#include <sstream>
```

```
#include <string>
```

Include dependency graph for `stream2string.h`:



This graph shows which files directly or indirectly include this file:



### 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

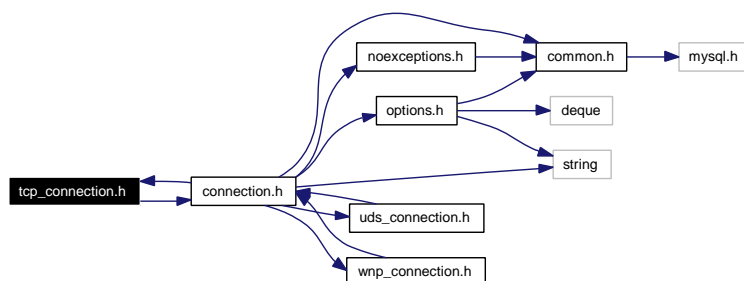
Declares an adapter that converts something that can be inserted into a C++ stream into a `std::string` type.

## 6.30 tcp\_connection.h File Reference

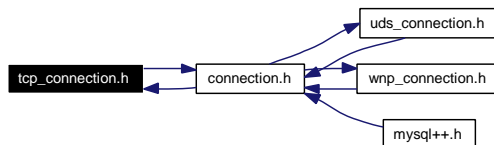
Declares the TCPConnection class.

```
#include "connection.h"
```

Include dependency graph for tcp\_connection.h:



This graph shows which files directly or indirectly include this file:



### Namespaces

- namespace **mysqlpp**

### Classes

- class **mysqlpp::TCPConnection**  
*Specialization of **Connection**(p. 31) for TCP/IP.*

#### 6.30.1 Detailed Description

Declares the TCPConnection class.

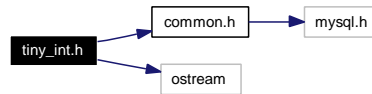
## 6.31 tiny\_int.h File Reference

Declares class for holding a SQL TINYINT.

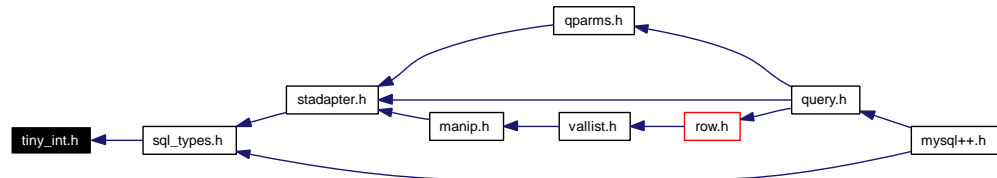
```
#include "common.h"
```

```
#include <ostream>
```

Include dependency graph for tiny\_int.h:



This graph shows which files directly or indirectly include this file:



## 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. 220) into a C++ stream.*

### **6.31.1 Detailed Description**

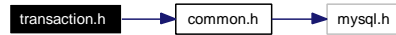
Declares class for holding a SQL TINYINT.

## 6.32 transaction.h File Reference

Declares the Transaction class.

```
#include "common.h"
```

Include dependency graph for transaction.h:



This graph shows which files directly or indirectly include this file:



### 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.

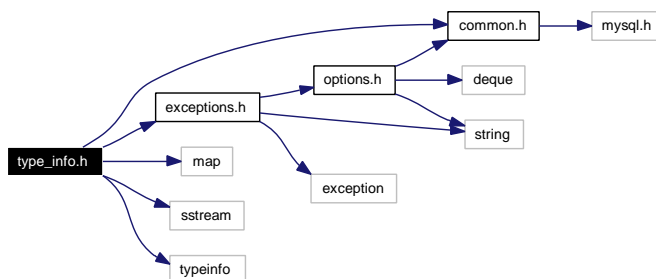


## 6.33 type\_info.h File Reference

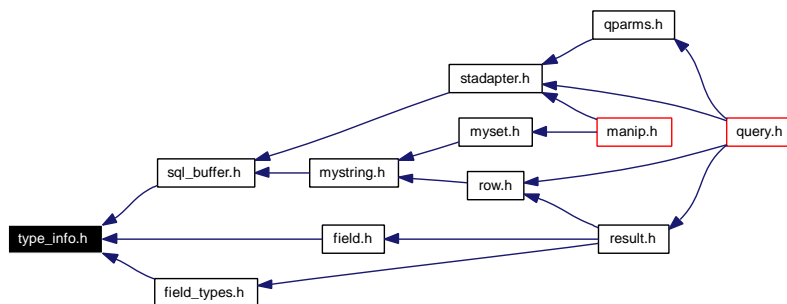
Declares classes that provide an interface between the SQL and C++ type systems.

```
#include "common.h"
#include "exceptions.h"
#include <map>
#include <sstream>
#include <typeinfo>
```

Include dependency graph for type\_info.h:



This graph shows which files directly or indirectly include this file:



### 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. 96) objects are equal.*

- **bool mysqlpp::operator!=** (const mysql\_type\_info &a, const mysql\_type\_info &b)

*Returns true if two **mysql\_type\_info**(p. 96) 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. 96) 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. 96) 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. 96) 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. 96) 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.

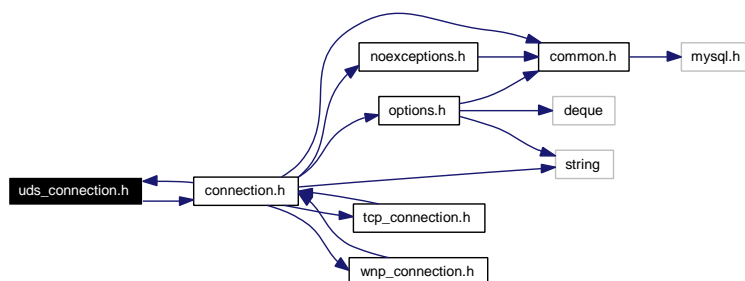
These classes are mostly used internal to the library.

## 6.34 uds\_connection.h File Reference

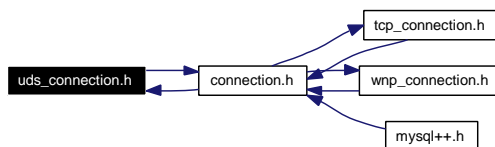
Declares the UnixDomainSocketConnection class.

```
#include "connection.h"
```

Include dependency graph for uds\_connection.h:



This graph shows which files directly or indirectly include this file:



### Namespaces

- namespace **mysqlpp**

### Classes

- class **mysqlpp::UnixDomainSocketConnection**  
*Specialization of **Connection**(p. 31) for Unix domain sockets.*

#### 6.34.1 Detailed Description

Declares the UnixDomainSocketConnection class.

## 6.35 vallist.h File Reference

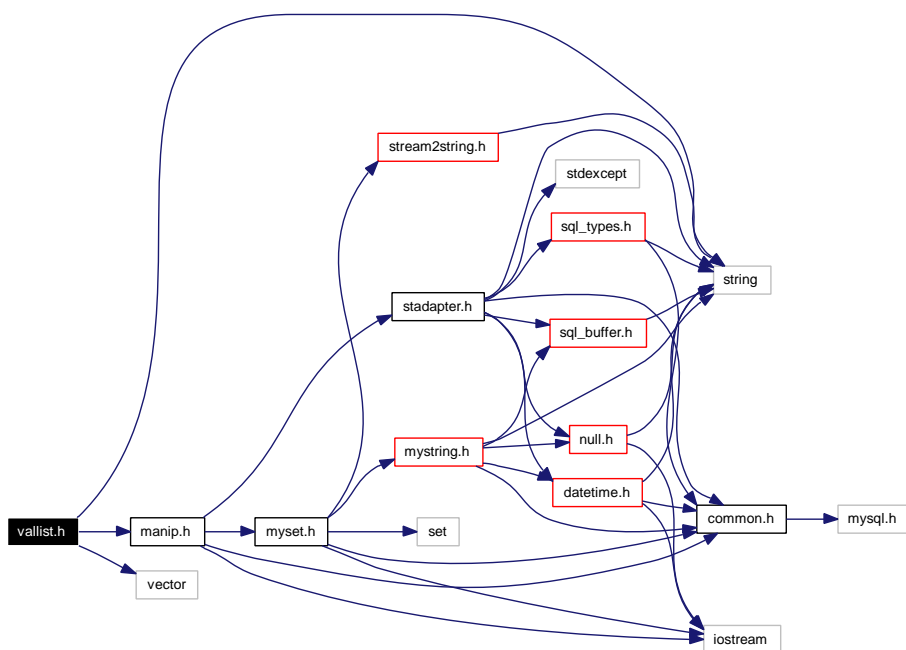
Declares templates for holding lists of values.

```
#include "manip.h"
```

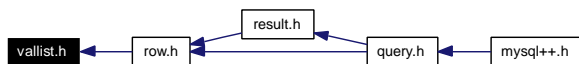
```
#include <string>
```

```
#include <vector>
```

Include dependency graph for vallist.h:



This graph shows which files directly or indirectly include this file:



## 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. 77), 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. 240), 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. 77) 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. 77), 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. 240) 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. 240), 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. 240).*

- 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. 240).*

- 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. 238) (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. 238) (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. 77).*
- `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. 77).*
- `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. 75) (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. 75) (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. 75) (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. 75) (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. 75) (sparse equal list).*

### 6.35.1 Detailed Description

Declares templates for holding lists of values.

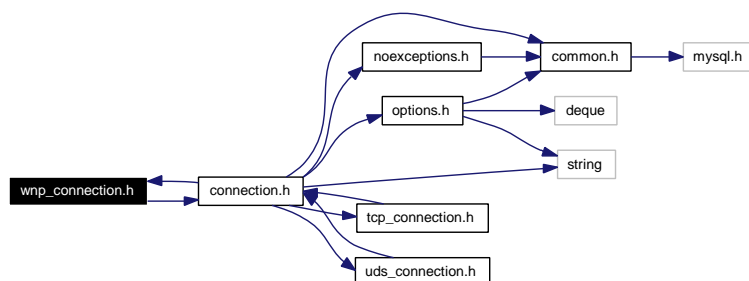


## 6.36 wnp\_connection.h File Reference

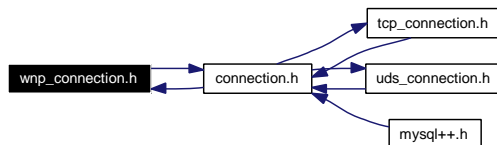
Declares the `WindowsNamedPipeConnection` class.

```
#include "connection.h"
```

Include dependency graph for `wnp_connection.h`:



This graph shows which files directly or indirectly include this file:



### Namespaces

- namespace `mysqlpp`

### Classes

- class `mysqlpp::WindowsNamedPipeConnection`  
*Specialization of `Connection`(p. 31) for Windows named pipes.*

#### 6.36.1 Detailed Description

Declares the `WindowsNamedPipeConnection` class.

# Index

- ~BeecryptMutex
  - mysqlpp::BeecryptMutex, 26
- ~Comparable
  - Comparable, 29
- ~ConnectionPool
  - mysqlpp::ConnectionPool, 44
- ~NoExceptions
  - mysqlpp::NoExceptions, 102
- ~RefCountedPointer
  - mysqlpp::RefCountedPointer, 147
- ~Transaction
  - mysqlpp::Transaction, 226
- affected\_rows
  - mysqlpp::DBDriver, 64
- assign
  - mysqlpp::RefCountedPointer, 148
  - mysqlpp::SQLTypeAdapter, 193
  - mysqlpp::String, 207
- at
  - mysqlpp::Row, 163
  - mysqlpp::SQLTypeAdapter, 193
  - mysqlpp::String, 207
- AutoFlag, 13
- autoflag.h, 247
- BadConversion
  - mysqlpp::BadConversion, 16
- BadFieldName
  - mysqlpp::BadFieldName, 19
- BadQuery
  - mysqlpp::BadQuery, 24
- base\_type
  - mysqlpp::mysql\_type\_info, 98
- BeecryptMutex
  - mysqlpp::BeecryptMutex, 26
- beemutex.h, 248
- before
  - mysqlpp::mysql\_type\_info, 98
- bound
  - mysqlpp::SQLQueryParms, 187
- c\_type
  - mysqlpp::mysql\_type\_info, 98
- clear
  - mysqlpp::ConnectionPool, 45
- client\_version
  - mysqlpp::DBDriver, 64
- commit
  - mysqlpp::Transaction, 226
- common.h, 250
- Comparable, 28
  - ~Comparable, 29
  - compare, 29
- comparable.h, 252
- compare
  - Comparable, 29
  - mysqlpp::Date, 52
  - mysqlpp::DateTime, 57
  - mysqlpp::SQLTypeAdapter, 194
  - mysqlpp::String, 207–209
  - mysqlpp::Time, 218
- connect
  - mysqlpp::Connection, 35
  - mysqlpp::DBDriver, 64
  - mysqlpp::TCPConnection, 214
  - mysqlpp::UnixDomainSocket-  
Connection, 231
  - mysqlpp::WindowsNamedPipe-  
Connection, 244
- connected
  - mysqlpp::Connection, 36
  - mysqlpp::DBDriver, 64
- Connection

- mysqlpp::Connection, 34, 35
- connection.h, 253
- ConnectionFailed
  - mysqlpp::ConnectionFailed, 42
- conv
  - mysqlpp::String, 209
- copy
  - mysqlpp::Connection, 36
  - mysqlpp::DBDriver, 64
- count\_rows
  - mysqlpp::Connection, 36
- cpool.h, 255
- create
  - mysqlpp::ConnectionPool, 45
- create\_db
  - mysqlpp::Connection, 36
  - mysqlpp::DBDriver, 65
- current\_field\_
  - mysqlpp::ResultBase, 155
- custom.h, 256
- data\_seek
  - mysqlpp::DBDriver, 65
- Date
  - mysqlpp::Date, 52
- DateTime
  - mysqlpp::DateTime, 56, 57
- datetime.h, 257
- DBDriver
  - mysqlpp::DBDriver, 63
- dbdriver.h, 259
- DBSelectionFailed
  - mysqlpp::DBSelectionFailed, 74
- destroy
  - mysqlpp::ConnectionPool, 45
- disconnect
  - mysqlpp::DBDriver, 65
- drop\_db
  - mysqlpp::Connection, 36
  - mysqlpp::DBDriver, 65
- enable\_ssl
  - mysqlpp::DBDriver, 65
- equal\_list
  - mysqlpp::Row, 163
- equal\_list\_b
  - mysqlpp::equal\_list\_b, 76
- equal\_list\_ba
  - mysqlpp::equal\_list\_ba, 78
- err\_api\_limit
  - mysqlpp::Option, 116
- err\_api\_reject
  - mysqlpp::Option, 116
- err\_connected
  - mysqlpp::Option, 116
- err\_NONE
  - mysqlpp::Option, 116
- errnum
  - mysqlpp::BadQuery, 24
  - mysqlpp::ConnectionFailed, 42
  - mysqlpp::DBDriver, 66
  - mysqlpp::DBSelectionFailed, 74
  - mysqlpp::Query, 125
- Error
  - mysqlpp::Option, 116
- error
  - mysqlpp::Connection, 37
  - mysqlpp::DBDriver, 66
  - mysqlpp::Query, 125
- escape\_q
  - mysqlpp::mysql\_type\_info, 98
- escape\_string
  - mysqlpp::DBDriver, 66
  - mysqlpp::Query, 125, 126
  - mysqlpp::SQLQueryParms, 187
- escape\_string\_no\_conn
  - mysqlpp::DBDriver, 66
- exceptions.h, 260
- exec
  - mysqlpp::Query, 127
- execute
  - mysqlpp::DBDriver, 66
  - mysqlpp::Query, 127, 128
- fetch\_field
  - mysqlpp::DBDriver, 66
- fetch\_lengths
  - mysqlpp::DBDriver, 67
- fetch\_row
  - mysqlpp::DBDriver, 67
  - mysqlpp::UseQueryResult, 235
- field.h, 263

- field\_list
  - mysqlpp::Row, 164, 165
- field\_names.h, 265
- field\_num
  - mysqlpp::ResultBase, 155
- field\_seek
  - mysqlpp::DBDriver, 67
  - mysqlpp::UseQueryResult, 235
- field\_types.h, 266
- for\_each
  - mysqlpp::Query, 129
- free\_result
  - mysqlpp::DBDriver, 67
- grab
  - mysqlpp::ConnectionPool, 45
- id
  - mysqlpp::mysql\_type\_info, 98
- insert
  - mysqlpp::Query, 130
- insert\_id
  - mysqlpp::DBDriver, 67
  - mysqlpp::Query, 130
- ipc\_info
  - mysqlpp::Connection, 37
  - mysqlpp::DBDriver, 68
- is\_null
  - mysqlpp::Null, 109
  - mysqlpp::SQLBuffer, 182
- is\_processed
  - mysqlpp::SQLTypeAdapter, 194
- is\_socket
  - mysqlpp::UnixDomainSocket-  
Connection, 231
- is\_wnp
  - mysqlpp::WindowsNamedPipe-  
Connection, 244
- iterator
  - mysqlpp::Row, 162
- kill
  - mysqlpp::Connection, 37
  - mysqlpp::DBDriver, 68
- length
  - mysqlpp::Field, 83
  - mysqlpp::SQLBuffer, 182
  - mysqlpp::String, 209
- list\_type
  - mysqlpp::Row, 162
- manip.h, 267
- max\_idle\_time
  - mysqlpp::ConnectionPool, 46
- max\_size
  - mysqlpp::String, 209
- more\_results
  - mysqlpp::DBDriver, 68
  - mysqlpp::Query, 130
- myset.h, 269
- mysql++.h, 271
  - MYSQLPP\_HEADER\_VERSION,  
272
  - MYSQLPP\_VERSION, 272
- mysql\_type\_info
  - mysqlpp::mysql\_type\_info, 97
- mysqlpp::BadConversion, 15
- mysqlpp::BadConversion
  - BadConversion, 16
- mysqlpp::BadFieldName, 18
- mysqlpp::BadFieldName
  - BadFieldName, 19
- mysqlpp::BadOption, 20
- mysqlpp::BadOption
  - what\_option, 21
- mysqlpp::BadParamCount, 22
- mysqlpp::BadQuery, 23
- mysqlpp::BadQuery
  - BadQuery, 24
  - errno, 24
- mysqlpp::BeecryptMutex, 26
- mysqlpp::BeecryptMutex
  - ~BeecryptMutex, 26
  - BeecryptMutex, 26
- mysqlpp::CompressOption, 30
- mysqlpp::Connection, 31
  - connect, 35
  - connected, 36
  - Connection, 34, 35
  - copy, 36
  - count\_rows, 36

- create\_db, 36
- drop\_db, 36
- error, 37
- ipc\_info, 37
- kill, 37
- operator private\_bool\_type, 37
- ping, 38
- query, 38
- select\_db, 38
- set\_option, 38
- thread\_id, 39
- thread\_start, 39
- mysqlpp::ConnectionFailed, 41
- mysqlpp::ConnectionFailed
  - ConnectionFailed, 42
  - errno, 42
- mysqlpp::ConnectionPool, 43
- mysqlpp::ConnectionPool
  - ~ConnectionPool, 44
  - clear, 45
  - create, 45
  - destroy, 45
  - grab, 45
  - max\_idle\_time, 46
  - release, 46
- mysqlpp::ConnectTimeoutOption, 47
- mysqlpp::DataOption, 48
- mysqlpp::Date, 50
  - compare, 52
  - Date, 52
  - operator time\_t, 52
  - year, 53
- mysqlpp::DateTime, 54
- mysqlpp::DateTime
  - compare, 57
  - DateTime, 56, 57
  - now, 57
  - year, 57, 58
- mysqlpp::DBDriver, 59
  - affected\_rows, 64
  - client\_version, 64
  - connect, 64
  - connected, 64
  - copy, 64
  - create\_db, 65
  - data\_seek, 65
  - DBDriver, 63
  - disconnect, 65
  - drop\_db, 65
  - enable\_ssl, 65
  - errno, 66
  - error, 66
  - escape\_string, 66
  - escape\_string\_no\_conn, 66
  - execute, 66
  - fetch\_field, 66
  - fetch\_lengths, 67
  - fetch\_row, 67
  - field\_seek, 67
  - free\_result, 67
  - insert\_id, 67
  - ipc\_info, 68
  - kill, 68
  - more\_results, 68
  - next\_result, 68
  - nr\_code, 63
  - nr\_error, 63
  - nr\_last\_result, 63
  - nr\_more\_results, 63
  - nr\_not\_supported, 63
  - num\_fields, 68
  - num\_rows, 69
  - ping, 69
  - protocol\_version, 69
  - query\_info, 69
  - refresh, 69
  - result\_empty, 70
  - server\_status, 70
  - server\_version, 70
  - set\_option, 70
  - shutdown, 71
  - store\_result, 71
  - thread\_aware, 71
  - thread\_end, 71
  - thread\_id, 71
  - thread\_start, 71
  - use\_result, 72
- mysqlpp::DBSelectionFailed, 73
- mysqlpp::DBSelectionFailed
  - DBSelectionFailed, 74
  - errno, 74
- mysqlpp::equal\_list\_b, 75

- equal\_list\_b, 76
- mysqlpp::equal\_list\_ba, 77
  - equal\_list\_ba, 78
- mysqlpp::Exception, 79
- mysqlpp::Field, 81
  - length, 83
- mysqlpp::FieldNames, 84
- mysqlpp::FieldTypes, 85
- mysqlpp::FieldTypes
  - operator=, 85
- mysqlpp::FoundRowsOption, 86
- mysqlpp::GuessConnectionOption, 87
- mysqlpp::IgnoreSpaceOption, 88
- mysqlpp::InitCommandOption, 89
- mysqlpp::InteractiveOption, 90
- mysqlpp::LocalFilesOption, 91
- mysqlpp::LocalInfileOption, 92
- mysqlpp::MultiResultsOption, 93
- mysqlpp::MultiStatementsOption, 94
- mysqlpp::MutexFailed, 95
- mysqlpp::mysql\_type\_info, 96
  - base\_type, 98
  - before, 98
  - c\_type, 98
  - escape\_q, 98
  - id, 98
  - mysql\_type\_info, 97
  - name, 99
  - operator=, 99
  - quote\_q, 99
  - sql\_name, 99
  - string\_type, 99
- mysqlpp::NamedPipeOption, 101
- mysqlpp::NoExceptions, 102
- mysqlpp::NoExceptions
  - ~NoExceptions, 102
  - NoExceptions, 102
- mysqlpp::NoSchemaOption, 104
- mysqlpp::Null, 105
  - is\_null, 109
  - Null, 107
  - operator Type &, 107
  - operator<, 108
  - operator=, 108
  - operator==, 108
- mysqlpp::null\_type, 110
- mysqlpp::NullIsBlank, 111
- mysqlpp::NullIsNull, 112
- mysqlpp::NullIsZero, 113
- mysqlpp::ObjectNotInitialized, 114
- mysqlpp::Option, 115
  - err\_api\_limit, 116
  - err\_api\_reject, 116
  - err\_connected, 116
  - err\_NONE, 116
  - Error, 116
- mysqlpp::OptionalExceptions, 117
- mysqlpp::OptionalExceptions
  - OptionalExceptions, 118
  - set\_exceptions, 118
- mysqlpp::Query, 119
  - errnum, 125
  - error, 125
  - escape\_string, 125, 126
  - exec, 127
  - execute, 127, 128
  - for\_each, 129
  - insert, 130
  - insert\_id, 130
  - more\_results, 130
  - operator void \*, 131
  - operator=, 131
  - parse, 131
  - Query, 125
  - replace, 131
  - reset, 132
  - store, 132, 133
  - store\_if, 133, 134
  - store\_next, 135
  - storein, 135
  - storein\_sequence, 136
  - storein\_set, 137
  - str, 137, 138
  - template\_defaults, 140
  - update, 138
  - use, 138, 139
- mysqlpp::ReadDefaultFileOption, 141
- mysqlpp::ReadDefaultGroupOption, 142
- mysqlpp::ReadTimeoutOption, 143
- mysqlpp::ReconnectOption, 144
- mysqlpp::RefCountedPointer, 145
- mysqlpp::RefCountedPointer

- ~RefCountedPointer, 147
- assign, 148
- operator const void \*, 148
- operator void \*, 148
- operator=, 149
- RefCountedPointer, 147
- mysqlpp::RefCountedPointerDestroyer, 150
- mysqlpp::RefCountedPointerDestroyer< MYSQL\_RES >, 151
- mysqlpp::ReportDataTruncationOption, 152
- mysqlpp::ResultBase, 153
- mysqlpp::ResultBase
  - current\_field\_, 155
  - field\_num, 155
- mysqlpp::Row, 157
  - at, 163
  - equal\_list, 163
  - field\_list, 164, 165
  - iterator, 162
  - list\_type, 162
  - operator private\_bool\_type, 166
  - operator[], 166
  - reference, 162
  - reverse\_iterator, 162
  - Row, 163
  - value\_list, 166–169
- mysqlpp::ScopedLock, 170
- mysqlpp::SecureAuthOption, 171
- mysqlpp::SelfTestFailed, 172
- mysqlpp::Set, 173
- mysqlpp::SetCharsetDirOption, 174
- mysqlpp::SetCharsetNameOption, 175
- mysqlpp::SetClientIpOption, 176
- mysqlpp::SharedMemoryBaseNameOption, 177
- mysqlpp::SimpleResult, 178
- mysqlpp::SimpleResult
  - operator private\_bool\_type, 179
- mysqlpp::SQLBuffer, 180
  - is\_null, 182
  - length, 182
  - SQLBuffer, 181
- mysqlpp::SQLParseElement, 183
- mysqlpp::SQLParseElement
  - SQLParseElement, 184
- mysqlpp::SQLQueryParms, 185
- mysqlpp::SQLQueryParms
  - bound, 187
  - escape\_string, 187
  - operator+, 187
  - set, 187
  - SQLQueryParms, 186
- mysqlpp::SQLTypeAdapter, 188
- mysqlpp::SQLTypeAdapter
  - assign, 193
  - at, 193
  - compare, 194
  - is\_processed, 194
  - operator=, 195
  - set\_processed, 195
  - SQLTypeAdapter, 192
  - type\_id, 195
- mysqlpp::SslOption, 196
- mysqlpp::SslOption
  - SslOption, 196
- mysqlpp::StoreQueryResult, 198
- mysqlpp::StoreQueryResult
  - operator private\_bool\_type, 199
- mysqlpp::String, 200
  - assign, 207
  - at, 207
  - compare, 207–209
  - conv, 209
  - length, 209
  - max\_size, 209
  - operator Null, 209
  - operator!=, 210
  - operator=, 210
  - operator==, 210
  - operator[], 211
  - size, 211
  - String, 205, 206
  - to\_string, 211
- mysqlpp::TCPConnection, 212
  - connect, 214
  - parse\_address, 214
  - TCPConnection, 213
- mysqlpp::Time, 216
  - compare, 218
  - operator time\_t, 218

- Time, 218
- mysqlpp::tiny\_int, 220
  - tiny\_int, 223
- mysqlpp::TooOld, 224
- mysqlpp::Transaction, 225
  - ~Transaction, 226
  - commit, 226
  - rollback, 226
  - Transaction, 225
- mysqlpp::TypeLookupFailed, 227
- mysqlpp::UnixDomainSocketConnection, 229
  - mysqlpp::UnixDomainSocketConnection
    - connect, 231
    - is\_socket, 231
    - UnixDomainSocketConnection, 230
- mysqlpp::UseEmbeddedConnectionOption, 232
  - mysqlpp::UseQueryError, 233
  - mysqlpp::UseQueryResult, 234
  - mysqlpp::UseQueryResult
    - fetch\_row, 235
    - field\_seek, 235
    - operator MYSQL\_RES \*, 236
  - mysqlpp::UseRemoteConnectionOption, 237
- mysqlpp::value\_list\_b, 238
  - value\_list\_b, 239
- mysqlpp::value\_list\_ba, 240
  - value\_list\_ba, 241
- mysqlpp::WindowsNamedPipeConnection, 242
  - mysqlpp::WindowsNamedPipe-
    - Connection
      - connect, 244
      - is\_wnp, 244
      - WindowsNamedPipeConnection, 243
- mysqlpp::WriteTimeoutOption, 245
- MYSQLPP\_HEADER\_VERSION
  - mysql++.h, 272
- MYSQLPP\_VERSION
  - mysql++.h, 272
- mystring.h, 274
- name
  - mysqlpp::mysql\_type\_info, 99
- next\_result
  - mysqlpp::DBDriver, 68
- NoExceptions
  - mysqlpp::NoExceptions, 102
- noexceptions.h, 277
- now
  - mysqlpp::DateTime, 57
- nr\_code
  - mysqlpp::DBDriver, 63
- nr\_error
  - mysqlpp::DBDriver, 63
- nr\_last\_result
  - mysqlpp::DBDriver, 63
- nr\_more\_results
  - mysqlpp::DBDriver, 63
- nr\_not\_supported
  - mysqlpp::DBDriver, 63
- Null
  - mysqlpp::Null, 107
- null.h, 279
- num\_fields
  - mysqlpp::DBDriver, 68
- num\_rows
  - mysqlpp::DBDriver, 69
- operator const void \*
  - mysqlpp::RefCountedPointer, 148
- operator MYSQL\_RES \*
  - mysqlpp::UseQueryResult, 236
- operator Null
  - mysqlpp::String, 209
- operator private\_bool\_type
  - mysqlpp::Connection, 37
  - mysqlpp::Row, 166
  - mysqlpp::SimpleResult, 179
  - mysqlpp::StoreQueryResult, 199
- operator time\_t
  - mysqlpp::Date, 52
  - mysqlpp::Time, 218
- operator Type &
  - mysqlpp::Null, 107
- operator void \*
  - mysqlpp::Query, 131
  - mysqlpp::RefCountedPointer, 148
- operator!=



- mysqlpp::String, 210
- operator+
  - mysqlpp::SQLQueryParms, 187
- operator<
  - mysqlpp::Null, 108
- operator=
  - mysqlpp::FieldTypes, 85
  - mysqlpp::mysql\_type\_info, 99
  - mysqlpp::Null, 108
  - mysqlpp::Query, 131
  - mysqlpp::RefCountedPointer, 149
  - mysqlpp::SQLTypeAdapter, 195
  - mysqlpp::String, 210
- operator==
  - mysqlpp::Null, 108
  - mysqlpp::String, 210
- operator[]
  - mysqlpp::Row, 166
  - mysqlpp::String, 211
- OptionalExceptions
  - mysqlpp::OptionalExceptions, 118
- options.h, 281
- parse
  - mysqlpp::Query, 131
- parse\_address
  - mysqlpp::TCPConnection, 214
- ping
  - mysqlpp::Connection, 38
  - mysqlpp::DBDriver, 69
- protocol\_version
  - mysqlpp::DBDriver, 69
- qparms.h, 285
- Query
  - mysqlpp::Query, 125
- query
  - mysqlpp::Connection, 38
- query.h, 287
- query\_info
  - mysqlpp::DBDriver, 69
- quote\_q
  - mysqlpp::mysql\_type\_info, 99
- refcounted.h, 290
- RefCountedPointer
  - mysqlpp::RefCountedPointer, 147
- reference
  - mysqlpp::Row, 162
- refresh
  - mysqlpp::DBDriver, 69
- release
  - mysqlpp::ConnectionPool, 46
- replace
  - mysqlpp::Query, 131
- reset
  - mysqlpp::Query, 132
- result.h, 291
- result\_empty
  - mysqlpp::DBDriver, 70
- reverse\_iterator
  - mysqlpp::Row, 162
- rollback
  - mysqlpp::Transaction, 226
- Row
  - mysqlpp::Row, 163
- row.h, 293
- select\_db
  - mysqlpp::Connection, 38
- server\_status
  - mysqlpp::DBDriver, 70
- server\_version
  - mysqlpp::DBDriver, 70
- set
  - mysqlpp::SQLQueryParms, 187
- set\_exceptions
  - mysqlpp::OptionalExceptions, 118
- set\_option
  - mysqlpp::Connection, 38
  - mysqlpp::DBDriver, 70
- set\_processed
  - mysqlpp::SQLTypeAdapter, 195
- shutdown
  - mysqlpp::DBDriver, 71
- size
  - mysqlpp::String, 211
- sql\_buffer.h, 295
- sql\_name
  - mysqlpp::mysql\_type\_info, 99
- sql\_types.h, 297
- SQLBuffer

- mysqlpp::SQLBuffer, 181
- SQLParseElement
  - mysqlpp::SQLParseElement, 184
- SQLQueryParms
  - mysqlpp::SQLQueryParms, 186
- SQLTypeAdapter
  - mysqlpp::SQLTypeAdapter, 192
- SslOption
  - mysqlpp::SslOption, 196
- stadapter.h, 298
- store
  - mysqlpp::Query, 132, 133
- store\_if
  - mysqlpp::Query, 133, 134
- store\_next
  - mysqlpp::Query, 135
- store\_result
  - mysqlpp::DBDriver, 71
- storein
  - mysqlpp::Query, 135
- storein\_sequence
  - mysqlpp::Query, 136
- storein\_set
  - mysqlpp::Query, 137
- str
  - mysqlpp::Query, 137, 138
- stream2string.h, 300
- String
  - mysqlpp::String, 205, 206
- string\_type
  - mysqlpp::mysql\_type\_info, 99
- tcp\_connection.h, 301
- TCPConnection
  - mysqlpp::TCPConnection, 213
- template\_defaults
  - mysqlpp::Query, 140
- thread\_aware
  - mysqlpp::DBDriver, 71
- thread\_end
  - mysqlpp::DBDriver, 71
- thread\_id
  - mysqlpp::Connection, 39
  - mysqlpp::DBDriver, 71
- thread\_start
  - mysqlpp::Connection, 39
- mysqlpp::DBDriver, 71
- Time
  - mysqlpp::Time, 218
- tiny\_int
  - mysqlpp::tiny\_int, 223
- tiny\_int.h, 302
- to\_string
  - mysqlpp::String, 211
- Transaction
  - mysqlpp::Transaction, 225
- transaction.h, 304
- type\_id
  - mysqlpp::SQLTypeAdapter, 195
- type\_info.h, 305
- uds\_connection.h, 307
- UnixDomainSocketConnection
  - mysqlpp::UnixDomainSocket-  
Connection, 230
- update
  - mysqlpp::Query, 138
- use
  - mysqlpp::Query, 138, 139
- use\_result
  - mysqlpp::DBDriver, 72
- vallist.h, 308
- value\_list
  - mysqlpp::Row, 166–169
- value\_list\_b
  - mysqlpp::value\_list\_b, 239
- value\_list\_ba
  - mysqlpp::value\_list\_ba, 241
- what\_option
  - mysqlpp::BadOption, 21
- WindowsNamedPipeConnection
  - mysqlpp::WindowsNamedPipe-  
Connection, 243
- wnp\_connection.h, 313
- year
  - mysqlpp::Date, 53
  - mysqlpp::DateTime, 57, 58