MySQL++ Reference Manual

1.7

Generated by Doxygen 1.2.18

Thu May 26 09:39:58 2005

Contents

Chapter 1

MySQL++ Reference Manual

1.0.1 Getting Started

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

1.0.2 Major Classes

In MySQL++, the main user-facing classes are **mysqlpp::Connection** (p. 42), **mysqlpp::Query** (p. 101), **mysqlpp::Result** (p. 108), and **mysqlpp::Row** (p. 116).

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.

1.0.3 Major Files

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

By the way, if, when installing this package, you didn't put the headers into their own subdirectory, you might consider reinstalling the package to remedy that. MySQL++ has a number of generically-named files (**convert.h,** (p. 161) **fields.h,** (p. 171) **row.h** (p. 186)...), so it's best to put them into a separate directory where they can't interfere with other code on your system. If you're on a Unixy platform, you do this by passing the `--includedir` option to the `configure` script. See the package's main README file for details.

1.0.4 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.

1.0.5 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++ Namespace Index

2.1 MySQL++ Namespace List

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

Chapter 3

MySQL++ Hierarchical Index

3.1 MySQL++ Class Hierarchy

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

Chapter 4

MySQL++ Compound Index

4.1 MySQL++ Compound List

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

Chapter 5

MySQL++ File Index

5.1 MySQL++ File List

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

Chapter 6

MySQL++ Namespace Documentation

6.1 mysqlpp Namespace Reference

Compounds

- class **BadConversion**

  Exception thrown when a bad type conversion is attempted.

- class **BadFieldName**

  Exception thrown when a requested named field doesn't exist.

- class **BadNullConversion**

  Exception thrown when you attempt to convert a SQL null to an incompatible type.

- class **BadQuery**

  Exception thrown when MySQL encounters a problem while processing your query.

- class **ColData_Tmpl**

  Template for string data that can convert itself to any standard C data type.

- class **Connection**

  Manages the connection to the MySQL database.

- class **const_string**

  Wrapper for `const char*` *to make it behave in a way more useful to MySQL++.*

- class **const_subscript_container**

  A base class that one derives from to become a random access container, which can be accessed with subscript notation.

- struct **cstr_equal_to**

  Function object that returns true if one const char is equal to another.*

- struct **cstr_greater**

  Function object that returns true if one const char is lexically "greater than" another.*

- struct **cstr_greater_equal**

  Function object that returns true if one const char is lexically "greater than or equal to" another.*

- struct **cstr_less**

  Function object that returns true if one const char is lexically "less than" another.*

- struct **cstr_less_equal**

  Function object that returns true if one const char is lexically "less than or equal to" another.*

- struct **cstr_not_equal_to**

  Function object that returns true if one const char is not equal to another.*

- struct **Date**

  Holds MySQL dates.

- struct **DateTime**

  A combination of the **Date** (p. 65) *and* **Time** (p. 144) *classes for holding MySQL DateTimes.*

- struct **DTbase**

  Base class template for MySQL++ date and time classes.

- struct **equal_list_b**

  Same as **equal_list_ba** (p. 74)*, plus the option to have some elements of the equals clause suppressed.*

- struct **equal_list_ba**

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

- class **FieldNames**

  Holds a list of SQL field names.

- class **Fields**

  A container similar to `std::vector` *for holding* **mysqlpp::Field** (p. 14) *records.*

- class **FieldTypes**

  A vector of SQL field types.

- struct **mysql_date**

  Base class of **Date** (p. 65).

- struct **mysql_dt_base**

  Base class for **mysql_date** (p. 80) *and* **mysql_time** (p. 83).

- struct **mysql_time**

  Base class of **Time** (p. 144).

- class **mysql_type_info**

Holds basic type information for ColData.

- class **MysqlCmp**

  Template for making function objects that can compare something against a **Row** (p. 116) *element.*

- class **MysqlCmpCStr**

  const char specialization of* **MysqlCmp** (p. 90)

- class **Null**

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

- class **null_type**

  The type of the global **mysqlpp::null** (p. 21) *object.*

- struct **NullisBlank**

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

- struct **NullisNull**

  Class for objects that define SQL null in terms of MySQL++'s **null_type** (p. 97).

- struct **NullisZero**

  Class for objects that define SQL null as 0.

- class **Query**

  A class for building and executing SQL queries.

- class **ResNSel**

  Holds the information on the success of queries that don't return any results.

- class **Result**

  This class manages SQL result sets.

- class **ResUse**

  A basic result set class, for use with "use" queries.

- class **Row**

  Manages rows from a result set.

- class **RowTemplate**

  Base class for class **Row** (p. 116).

- class **Set**

  A special std::set derivative for holding MySQL data sets.

- struct **SQLParseElement**

  Used within **SQLQuery** (p. 132) *to hold elements for parameterized queries.*

- class **SQLQuery**

  The base class for **mysqlpp::Query** (p. 101).

- class **SQLQueryNEParms**
  Exception thrown when not enough parameters are provided.
- class **SQLQueryParms**
  This class holds the parameter values for filling template queries.
- class **SQLString**
  A specialized `std::string` *that will convert from any valid MySQL type.*
- class **subscript_iterator**
  Iterator that can be subscripted.
- struct **Time**
  Holds MySQL times.
- class **tiny_int**
  Class for holding an SQL `tiny_int` *object.*
- struct **value_list_b**
  Same as **value_list_ba** (p. 151)*, plus the option to have some elements of the list suppressed.*
- struct **value_list_ba**
  Holds a list of items, typically used to construct a SQL "value list".

Typedefs

- typedef **ColData_Tmpl**< **const_string** > **ColData**
  The type that is returned by constant rows.
- typedef **ColData_Tmpl**< std::string > **MutableColData**
  The type that is returned by mutable rows.
- typedef std::binary_function< const char *, const char *, bool > **bin_char_pred**
  Base class for the other predicate types defined in **compare.h**.
- typedef unsigned long long **ulonglong**
  unsigned 64-bit integer type for GCC-based systems
- typedef long long **longlong**
  signed 64-bit integer type for GCC-based systems
- typedef MYSQL_FIELD **Field**
  Alias for MYSQL_FIELD.
- typedef const char **cchar**
  Contraction for 'const char'.*
- typedef unsigned int **uint**
  Contraction for 'unsigned int'.

Enumerations

- enum **sql_cmp_type**

  Appears to be unused! Remove?

- enum **quote_type0** { **quote** }
- enum **quote_only_type0** { **quote_only** }
- enum **quote_double_only_type0** { **quote_double_only** }
- enum **escape_type0**
- enum **do_nothing_type0** { **do_nothing** }
- enum **ignore_type0** { **ignore** }
- enum **query_reset**

  Used for indicating whether a query object should auto-reset or not.

Functions

- template<class BinaryPred, class CmpType> **MysqlCmp**< BinaryPred, CmpType > **mysql_cmp** (**uint** i, const BinaryPred &func, const CmpType &cmp2)

  Template for function objects that compare any two objects, as long as they can be converted to **SQLString** (p. 140).

- template<class BinaryPred> **MysqlCmpCStr**< BinaryPred > **mysql_cmp_cstr** (**uint** i, const BinaryPred &func, const char *cmp2)

  Template for function objects that compare any two things that can be converted to `const char*`.

- std::ostream & **operator**<< (std::ostream &o, const **const_string** &str)

  Inserts a **const_string** (p. 54) *into a C++ stream.*

- int **compare** (const **const_string** &lhs, const **const_string** &rhs)

  Calls lhs.compare(), passing rhs.

- bool **operator**== (**const_string** &lhs, **const_string** &rhs)

  Returns true if lhs is the same as rhs.

- bool **operator!**= (**const_string** &lhs, **const_string** &rhs)

  Returns true if lhs is not the same as rhs.

- bool **operator**< (**const_string** &lhs, **const_string** &rhs)

  Returns true if lhs is lexically less than rhs.

- bool **operator**<= (**const_string** &lhs, **const_string** &rhs)

  Returns true if lhs is lexically less or equal to rhs.

- bool **operator**> (**const_string** &lhs, **const_string** &rhs)

  Returns true if lhs is lexically greater than rhs.

- bool **operator**>= (**const_string** &lhs, **const_string** &rhs)

  Returns true if lhs is lexically greater than or equal to rhs.

- std::ostream & **operator**<< (std::ostream &s, const **Date** &d)

  Inserts a **Date** (p. 65) *object into a C++ stream in a MySQL-compatible format.*

- std::ostream & **operator**<< (std::ostream &s, const **Time** &d)

  Inserts a **Time** (p. 144) *object into a C++ stream in a MySQL-compatible format.*

- std::ostream & **operator**<< (std::ostream &s, const **DateTime** &d)

  Inserts a **DateTime** (p. 67) *object into a C++ stream in a MySQL-compatible format.*

- **SQLQueryParms** & **operator**<< (quote_type2 p, **SQLString** &in)

  Inserts a **SQLString** (p. 140) *into a stream, quoted and escaped.*

- template<> ostream & **operator**<< (quote_type1 o, const string &in)

  Inserts a C++ string into a stream, quoted and escaped.

- template<> ostream & **operator**<< (quote_type1 o, const char *const &in)

  Inserts a C string into a stream, quoted and escaped.

- template<class Str> ostream & **_manip** (quote_type1 o, const **ColData_Tmpl**< Str > &in)

  Utility function used by operator<<(quote_type1, ColData).

- template<> ostream & **operator**<< (quote_type1 o, const **ColData_Tmpl**< string > &in)

  Inserts a ColData into a stream, quoted and escaped.

- template<> ostream & **operator**<< (quote_type1 o, const **ColData_Tmpl**< **const_string** > &in)

  Inserts a ColData with const string into a stream, quoted and escaped.

- ostream & **operator**<< (ostream &o, const **ColData_Tmpl**< string > &in)

  Inserts a ColData into a stream.

- ostream & **operator**<< (ostream &o, const **ColData_Tmpl**< **const_string** > &in)

  Inserts a ColData with const string into a stream.

- **SQLQuery** & **operator**<< (**SQLQuery** &o, const **ColData_Tmpl**< string > &in)

  Insert a ColData into a **SQLQuery** (p. 132).

- **SQLQuery** & **operator**<< (**SQLQuery** &o, const **ColData_Tmpl**< **const_string** > &in)

  Insert a ColData with const string into a **SQLQuery** (p. 132).

- **SQLQueryParms** & **operator**<< (quote_only_type2 p, **SQLString** &in)

  Inserts a **SQLString** (p. 140) *into a stream, quoting it unless it's data that needs no quoting.*

- template<> ostream & **operator**<< (quote_only_type1 o, const **ColData_Tmpl**< string > &in)

  Inserts a ColData into a stream, quoted.

- template<> ostream & **operator**<< (quote_only_type1 o, const **ColData_Tmpl**< **const_string** > &in)

  Inserts a ColData with const string into a stream, quoted.

- **SQLQueryParms** & **operator**<< (quote_double_only_type2 p, **SQLString** &in)

  Inserts a **SQLString** (p. 140) *into a stream, double-quoting it (") unless it's data that needs no quoting.*

- template<> ostream & **operator**<< (quote_double_only_type1 o, const **ColData_Tmpl**< string > &in)

  Inserts a ColData into a stream, double-quoted (").

- template<> ostream & **operator**<< (quote_double_only_type1 o, const **ColData_Tmpl**< **const_string** > &in)

  Inserts a ColData with const string into a stream, double-quoted (").

- **SQLQueryParms** & **operator**<< (escape_type2 p, **SQLString** &in)

  Inserts a **SQLString** (p. 140) *into a stream, escaping special SQL characters.*

- template<> std::ostream & **operator**<< (escape_type1 o, const std::string &in)

  Inserts a C++ string into a stream, escaping special SQL characters.

- template<> ostream & **operator**<< (escape_type1 o, const char *const &in)

  Inserts a C string into a stream, escaping special SQL characters.

- template<class Str> ostream & **_manip** (escape_type1 o, const **ColData_Tmpl**< Str > &in)

  Utility function used by operator<<(escape_type1, ColData).

- template<> std::ostream & **operator**<< (escape_type1 o, const **ColData_Tmpl**< std::string > &in)

  Inserts a ColData into a stream, escaping special SQL characters.

- template<> std::ostream & **operator**<< (escape_type1 o, const **ColData_Tmpl**< **const_string** > &in)

  Inserts a ColData with const string into a stream, escaping special SQL characters.

- template<class T> std::ostream & **operator**<< (escape_type1 o, const T &in)

  Inserts any type T into a stream that has an operator<< defined for it.

- template<> std::ostream & **operator**<< (escape_type1 o, char *const &in)

  Inserts a C string into a stream, escaping special SQL characters.

- template<class Container> std::ostream & **operator**<< (std::ostream &s, const **Set**< Container > &d)

  Inserts a **Set** (p. 129) *object into a C++ stream.*

- template<class Type, class Behavior> std::ostream & **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.

- void **swap** (**ResUse** &x, **ResUse** &y)

  Swaps two **ResUse** (p. 110) *objects.*

- void **swap** (**Result** &x, **Result** &y)

  Swaps two **Result** (p. 108) *objects.*

- template<class Strng, class T> Strng **stream2string** (const T &object)

  Converts a stream-able object to any type that can be initialized from an `std::string`.

- void **strip** (std::string &s)

  Strips blanks at left and right ends.

- void **escape_string** (std::string &s)

  C++ equivalent of mysql_escape_string().

- void **str_to_upr** (std::string &s)

  Changes case of string to upper.

- void **str_to_lwr** (std::string &s)

  Changes case of string to lower.

- void **strip_all_blanks** (std::string &s)

  Removes all blanks.

- void **strip_all_non_num** (std::string &s)

  Removes all non-numerics.

- bool **operator==** (const **mysql_type_info** &a, const **mysql_type_info** &b)

  Returns true if two **mysql_type_info** (p. 85) *objects are equal.*

- bool **operator!=** (const **mysql_type_info** &a, const **mysql_type_info** &b)

  Returns true if two **mysql_type_info** (p. 85) *objects are not equal.*

- bool **operator==** (const std::type_info &a, const **mysql_type_info** &b)

  Returns true if a given **mysql_type_info** (p. 85) *object is equal to a given C++ type_info object.*

- bool **operator!=** (const std::type_info &a, const **mysql_type_info** &b)

  Returns true if a given **mysql_type_info** (p. 85) *object is not equal to a given C++ type_info object.*

- bool **operator==** (const **mysql_type_info** &a, const std::type_info &b)

  Returns true if a given **mysql_type_info** (p. 85) *object is equal to a given C++ type_info object.*

- bool **operator!=** (const **mysql_type_info** &a, const std::type_info &b)

  Returns true if a given **mysql_type_info** (p. 85) *object is not equal to a given C++ type_info object.*

- void **create_vector** (int size, std::vector< bool > &v, bool t0, bool t1, bool t2, bool t3, bool t4, bool t5, bool t6, bool t7, bool t8, bool t9, bool ta, bool tb, bool tc)

Create a vector of bool with the given arguments as values.

- template<class Container> void **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 Seq1, class Seq2, class Manip> std::ostream & **operator<<** (std::ostream &o, const **equal_list_ba**< Seq1, Seq2, Manip > &el)

  Inserts an **equal_list_ba** (p. 74) *into an std::ostream.*

- template<class Seq1, class Seq2, class Manip> std::ostream & **operator<<** (std::ostream &o, const **equal_list_b**< Seq1, Seq2, Manip > &el)

  Same as operator<< for **equal_list_ba** (p. 74)*, plus the option to suppress insertion of some list items in the stream.*

- template<class Seq, class Manip> std::ostream & **operator<<** (std::ostream &o, const **value_list_ba**< Seq, Manip > &cl)

  Inserts a **value_list_ba** (p. 151) *into an std::ostream.*

- template<class Seq, class Manip> std::ostream & **operator<<** (std::ostream &o, const **value_list_b**< Seq, Manip > &cl)

  Same as operator<< for **value_list_ba** (p. 151)*, plus the option to suppress insertion of some list items in the stream.*

- template<class Seq> **value_list_ba**< Seq, **do_nothing_type0** > **value_list** (const Seq &s, const char *d=",")

  Constructs a **value_list_ba** (p. 151)*.*

- template<class Seq, class Manip> **value_list_ba**< Seq, Manip > **value_list** (const Seq &s, const char *d, Manip m)

  Constructs a **value_list_ba** (p. 151)*.*

- template<class Seq, class Manip> **value_list_b**< Seq, Manip > **value_list** (const Seq &s, const char *d, Manip m, const std::vector< bool > &vb)

  Constructs a **value_list_b** (p. 149) *(sparse value list).*

- template<class Seq, class Manip> **value_list_b**< Seq, Manip > **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. 149) *(sparse value list).*

- template<class Seq> **value_list_b**< Seq, **do_nothing_type0** > **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** > **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** > **equal_list** (const Seq1 &s1, const Seq2 &s2, const char *d=",", const char *e="=")

  Constructs an **equal_list_ba** (p. 74).

- template<class Seq1, class Seq2, class Manip> **equal_list_ba**< Seq1, Seq2, Manip > **equal_list** (const Seq1 &s1, const Seq2 &s2, const char *d, const char *e, Manip m)

  Constructs an **equal_list_ba** (p. 74).

- template<class Seq1, class Seq2, class Manip> **equal_list_b**< Seq1, Seq2, Manip > **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. 72) *(sparse equal list).*

- template<class Seq1, class Seq2, class Manip> **equal_list_b**< Seq1, Seq2, Manip > **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. 72) *(sparse equal list).*

- template<class Seq1, class Seq2> **equal_list_b**< Seq1, Seq2, **do_nothing_type0** > **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. 72) *(sparse equal list).*

- template<class Seq1, class Seq2> **equal_list_b**< Seq1, Seq2, **do_nothing_type0** > **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. 72) *(sparse equal list).*

- template<class Seq1, class Seq2> **equal_list_b**< Seq1, Seq2, **do_nothing_type0** > **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. 72) *(sparse equal list).*

Variables

- const bool **use_exceptions** = true

  Alias for 'true', to make code requesting exceptions more readable.

- bool **dont_quote_auto** = false

  Set (p. 129) *to true if you want to suppress automatic quoting.*

- const **null_type null** = **null_type**()

  Global 'null' instance. Use wherever you need a SQL null. (As opposed to a C++ language null pointer or null character.).

6.1.1 Detailed Description

All global symbols in MySQL++ are in namespace mysqlpp. This is needed because many symbols are rather generic (e.g. **Row** (p. 116), **Query** (p. 101)...), so there is a serious danger of conflicts.

6.1.2 Enumeration Type Documentation

6.1.2.1 enum mysqlpp::do_nothing_type0

The 'do_nothing' manipulator.

Does exactly what it says: nothing. Used as a dummy manipulator when you are required to use some manipulator but don't want anything to be done to the following item. When used with **SQLQueryParms** (p. 138) it will make sure that it does not get formatted in any way, overriding any setting set by the template query.

Enumeration values:

do_nothing insert into a std::ostream to override manipulation of next item

6.1.2.2 enum mysqlpp::escape_type0

The 'escape' manipulator.

Calls mysql_escape_string() in the MySQL C API on the following argument to prevent any special SQL characters from being interpreted.

6.1.2.3 enum mysqlpp::ignore_type0

The 'ignore' manipulator.

Only valid when used with **SQLQueryParms** (p. 138). It's a dummy manipulator like the `do_nothing manipulator`, except that it will not override formatting set by the template query. It is simply ignored.

Enumeration values:

ignore insert into a std::ostream as a dummy manipulator

6.1.2.4 enum mysqlpp::quote_double_only_type0

The 'double_quote_only' manipulator.

Similar to `quote_only manipulator`, except that it uses double quotes instead of single quotes.

Enumeration values:

quote_double_only insert into a std::ostream to double-quote next item

6.1.2.5 enum mysqlpp::quote_only_type0

The 'quote_only' manipulator.

Similar to `quote manipulator`, except that it doesn't escape special SQL characters.

Enumeration values:

quote_only insert into a std::ostream to single-quote next item

6.1.2.6 enum mysqlpp::quote_type0

The standard 'quote' manipulator.

Insert this into a stream to put single quotes around the next item in the stream, and escape characters within it that are 'special' in SQL. This is the most generally useful of the manipulators.

Enumeration values:

quote insert into a std::ostream to single-quote and escape next item

6.1.3 Function Documentation

6.1.3.1 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.

This function is used with the **ResUse** (p. 110) and **Result** (p. 108) containers, which have a field_num() member function that maps a field name to its position number. So for each named field, we set the bool in the vector at the corresponding position to true.

This function is used within the library to build the vector used in calling the vector form of **Row::equal_list()** (p. 122), **Row::value_list()** (p. 128), and **Row::field_list()** (p. 125). See the "Harnessing SSQLS Internals" section of the user manual to see that feature at work.

6.1.3.2 void mysqlpp::create_vector (int *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.

This function takes up to 13 bools, with the size parameter controlling the actual number of parameters we pay attention to.

This function is used within the library to build the vector used in calling the vector form of **Row::equal_list()** (p. 122), **Row::value_list()** (p. 128), and **Row::field_list()** (p. 125). See the "Harnessing SSQLS Internals" section of the user manual to see that feature at work.

6.1.3.3 template<class Seq1, class Seq2> equal_list_b<Seq1, Seq2, do_nothing_type0> 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. 72) (sparse equal list).

Same as equal_list(Seq&, Seq&, const char*, bool, bool...) except that it doesn't take the const char* argument. It uses a comma for the delimiter. This form is useful for building simple equals lists, where no manipulators are necessary, and the default delimiter and equals symbol are suitable.

6.1.3.4 template<class Seq1, class Seq2> equal_list_b<Seq1, Seq2, do_nothing_type0> 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. 72) (sparse equal list).

Same as equal_list(Seq&, Seq&, const char*, const char*, bool, bool...) except that it doesn't take the second const char* argument. It uses " = " for the equals symbol.

6.1.3.5 template<class Seq1, class Seq2> equal_list_b<Seq1, Seq2, do_nothing_type0> 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. 72) (sparse equal list).

Same as equal_list(Seq&, Seq&, const char*, const char*, Manip, bool, bool...) except that it doesn't take the Manip argument. It uses the do_nothing manipulator instead, meaning that none of the elements are escaped when being inserted into a stream.

6.1.3.6 template<class Seq1, class Seq2, class Manip> equal_list_b<Seq1, Seq2, Manip> 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. 72) (sparse equal list).

Same as equal_list(Seq&, Seq&, const char*, const char*, Manip, vector<bool>&) except that it takes boolean parameters instead of a list of bools.

6.1.3.7 template<class Seq1, class Seq2, class Manip> equal_list_b<Seq1, Seq2, Manip> 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. 72) (sparse equal list).

Same as equal_list(Seq&, Seq&, const char*, const char*, Manip) except that you can pass a vector of bools. For each true item in that list, operator<< adds the corresponding item is put in the equal list. This lets you pass in sequences when you don't want all of the elements to be inserted into a stream.

6.1.3.8 template<class Seq1, class Seq2, class Manip> equal_list_ba<Seq1, Seq2, Manip> equal_list (const Seq1 & *s1*, const Seq2 & *s2*, const char * *d*, const char * *e*, Manip *m*)

Constructs an **equal_list_ba** (p. 74).

Same as equal_list(Seq&, Seq&, const char*, const char*) except that it also lets you specify the manipulator. Use this version if the data must be escaped or quoted when being inserted into a stream.

6.1.3.9 template<class Seq1, class Seq2> equal_list_ba<Seq1, Seq2, do_nothing_type0> equal_list (const Seq1 & *s1*, const Seq2 & *s2*, const char * *d* = ",", const char * *e* = " = ")

Constructs an **equal_list_ba** (p. 74).

This function returns an equal list that uses the 'do_nothing' manipulator. That is, the items are not quoted or escaped in any way when inserted into a stream. See equal_list(Seq, Seq, const char*, const char*, Manip) if you need a different manipulator.

The idea is for both lists to be of equal length because corresponding elements from each list are handled as pairs, but if one list is shorter than the other, the generated list will have that many elements.

Parameters:

s1 items on the left side of the equals sign when the equal list is inserted into a stream

s2 items on the right side of the equals sign

d delimiter operator<< should place between pairs

e what operator<< should place between items in each pair; by default, an equals sign, as that is the primary use for this mechanism.

6.1.3.10 template<class BinaryPred, class CmpType> MysqlCmp<BinaryPred, CmpType> mysql_cmp (uint *i*, const BinaryPred & *func*, const CmpType & *cmp2*)

Template for function objects that compare any two objects, as long as they can be converted to **SQLString** (p. 140).

This is a more generic form of **mysql_cmp_cstr()** (p. 25), and is therefore less efficient. Use this form only when necessary.

Parameters:

i field index number

func one of the functors in **compare.h**, or any compatible functor

cmp2 what to compare to

6.1.3.11 template<class BinaryPred> MysqlCmpCStr<BinaryPred> mysql_cmp_cstr (uint *i*, const BinaryPred & *func*, const char * *cmp2*)

Template for function objects that compare any two things that can be converted to `const char*`.

Parameters:

i field index number

func one of **cstr_equal_to** (p. 59), **cstr_not_equal_to** (p. 64), **cstr_less** (p. 62), **cstr_less_equal** (p. 63), **cstr_less_equal** (p. 63), or **cstr_less_equal** (p. 63).

cmp2 what to compare to

See also:

mysql_cmp() (p. 24)

6.1.3.12 template<class Seq, class Manip> std::ostream& operator<< (std::ostream & *o*, const value_list_b< Seq, Manip > & *cl*)

Same as operator<< for **value_list_ba** (p. 151), plus the option to suppress insertion of some list items in the stream.

See **value_list_b** (p. 149)'s documentation for examples of how this works.

6.1.3.13 template<class Seq, class Manip> std::ostream& operator<< (std::ostream & *o*, const value_list_ba< Seq, Manip > & *cl*)

Inserts a **value_list_ba** (p. 151) into an std::ostream.

Given a list (a, b) and a delimiter D, this operator will insert "aDb" into the stream.

See **value_list_ba** (p. 151)'s documentation for concrete examples.

See also:

value_list() (p. 30)

6.1.3.14 template<class Seq1, class Seq2, class Manip> std::ostream& operator<< (std::ostream & *o*, const equal_list_b< Seq1, Seq2, Manip > & *el*)

Same as operator<< for **equal_list_ba** (p. 74), plus the option to suppress insertion of some list items in the stream.

See **equal_list_b** (p. 72)'s documentation for examples of how this works.

6.1.3.15 template<class Seq1, class Seq2, class Manip> std::ostream& operator<< (std::ostream & *o*, const equal_list_ba< Seq1, Seq2, Manip > & *el*)

Inserts an **equal_list_ba** (p. 74) into an std::ostream.

Given two lists (a, b) and (c, d), a delimiter D, and an equals symbol E, this operator will insert "aEcDbEd" into the stream.

See **equal_list_ba** (p. 74)'s documentation for concrete examples.

See also:

equal_list() (p. 24)

6.1.3.16 template<> std::ostream& operator<< (escape_type1 *o*, char *const & *in*) [inline]

Inserts a C string into a stream, escaping special SQL characters.

This version exists solely to handle constness problems. We force everything to the completely-const version: operator<<(escape_type1, const char* const&).

6.1.3.17 template<class T> std::ostream& operator<< (escape_type1 *o*, const T & *in*) [inline]

Inserts any type T into a stream that has an operator<< defined for it.

Does not actually escape that data! Use one of the other forms of operator<< for the escape manipulator if you need escaping. This template exists to catch cases like inserting an `int` after the escape manipulator: you don't actually want escaping in this instance.

6.1.3.18 template<> std::ostream & mysqlpp::operator<< (escape_type1 *o*, const ColData_Tmpl< const_string > & *in*)

Inserts a ColData with const string into a stream, escaping special SQL characters.

Because ColData was designed to contain MySQL type data, we may choose not to escape the data, if it is not needed.

6.1.3.19 template<> std::ostream & mysqlpp::operator<< (escape_type1 *o*, const ColData_Tmpl< std::string > & *in*)

Inserts a ColData into a stream, escaping special SQL characters.

Because ColData was designed to contain MySQL type data, we may choose not to escape the data, if it is not needed.

6.1.3.20 template<> std::ostream & mysqlpp::operator<< (escape_type1 *o*, const char *const & *in*)

Inserts a C string into a stream, escaping special SQL characters.

Because C's type system lacks the information we need to second- guess this manipulator, we always run the escaping algorithm on the data, even if it's not needed.

6.1.3.21 template<> std::ostream & mysqlpp::operator<< (escape_type1 *o*, const std::string & *in*)

Inserts a C++ string into a stream, escaping special SQL characters.

Because std::string lacks the type information we need, the string is always escaped, even if it doesn't need it.

6.1.3.22 SQLQueryParms & mysqlpp::operator<< (escape_type2 *p*, SQLString & *in*)

Inserts a **SQLString** (p. 140) into a stream, escaping special SQL characters.

We actually only do the escaping if in.is_string is set but in.dont_escape is not. If that is not the case, we insert the string data directly.

6.1.3.23 template<> ostream& operator<< (quote_double_only_type1 *o*, const ColData_Tmpl< const_string > & *in*)

Inserts a ColData with const string into a stream, double-quoted (").

Because ColData was designed to contain MySQL type data, we may choose not to actually quote the data, if it is not needed.

6.1.3.24 template<> ostream& operator<< (quote_double_only_type1 *o*, const ColData_Tmpl< string > & *in*)

Inserts a ColData into a stream, double-quoted (").

Because ColData was designed to contain MySQL type data, we may choose not to actually quote the data, if it is not needed.

6.1.3.25 SQLQueryParms& operator<< (quote_double_only_type2 *p*, SQLString & *in*)

Inserts a **SQLString** (p. 140) into a stream, double-quoting it (") unless it's data that needs no quoting.

We make the decision to quote the data based on the in.is_string flag. You can set it yourself, but **SQLString** (p. 140)'s ctors should set it correctly for you.

6.1.3.26 template<> ostream& operator<< (quote_only_type1 *o*, const ColData_Tmpl< const_string > & *in*)

Inserts a ColData with const string into a stream, quoted.

Because ColData was designed to contain MySQL type data, we may choose not to actually quote the data, if it is not needed.

6.1.3.27 template<> ostream& operator<< (quote_only_type1 *o*, const ColData_Tmpl< string > & *in*)

Inserts a ColData into a stream, quoted.

Because ColData was designed to contain MySQL type data, we may choose not to actually quote the data, if it is not needed.

6.1.3.28 SQLQueryParms& operator<< (quote_only_type2 *p*, SQLString & *in*)

Inserts a **SQLString** (p. 140) into a stream, quoting it unless it's data that needs no quoting.

We make the decision to quote the data based on the in.is_string flag. You can set it yourself, but **SQLString** (p. 140)'s ctors should set it correctly for you.

6.1.3.29 SQLQuery& operator<< (SQLQuery & *o*, const ColData_Tmpl< const_string > & *in*)

Insert a ColData with const string into a **SQLQuery** (p. 132).

This operator appears to be a workaround for a weakness in one compiler's implementation of the C++ type system. See Wishlist for current plan on what to do about this.

6.1.3.30 SQLQuery& operator<< (SQLQuery & *o*, const ColData_Tmpl< string > & *in*)

Insert a ColData into a **SQLQuery** (p. 132).

This operator appears to be a workaround for a weakness in one compiler's implementation of the C++ type system. See Wishlist for current plan on what to do about this.

6.1.3.31 ostream& operator<< (ostream & *o*, const ColData_Tmpl< const_string > & *in*)

Inserts a ColData with const string into a stream.

Because ColData was designed to contain MySQL type data, this operator has the information needed to choose to quote and/or escape the data as it is inserted into the stream, even if you don't use any of the quoting or escaping manipulators.

6.1.3.32 ostream& operator<< (ostream & *o*, const ColData_Tmpl< string > & *in*)

Inserts a ColData into a stream.

Because ColData was designed to contain MySQL type data, this operator has the information needed to choose to quote and/or escape the data as it is inserted into the stream, even if you don't use any of the quoting or escaping manipulators.

6.1.3.33 template<> ostream& operator<< (quote_type1 *o*, const ColData_Tmpl< const_string > & *in*)

Inserts a ColData with const string into a stream, quoted and escaped.

Because ColData was designed to contain MySQL type data, we may choose not to actually quote or escape the data, if it is not needed.

6.1.3.34 template<> ostream& operator<< (quote_type1 *o*, const ColData_Tmpl< string > & *in*)

Inserts a ColData into a stream, quoted and escaped.

Because ColData was designed to contain MySQL type data, we may choose not to actually quote or escape the data, if it is not needed.

6.1.3.35 template<> ostream& operator<< (quote_type1 *o*, const char *const & *in*)

Inserts a C string into a stream, quoted and escaped.

Because C strings lack the type information we need, the string is both quoted and escaped, always.

6.1.3.36 template<> ostream& operator<< (quote_type1 *o*, const string & *in*)

Inserts a C++ string into a stream, quoted and escaped.

Because std::string lacks the type information we need, the string is both quoted and escaped, always.

6.1.3.37 SQLQueryParms& operator<< (quote_type2 *p*, SQLString & *in*)

Inserts a **SQLString** (p. 140) into a stream, quoted and escaped.

If in.is_string is set and in.dont_escape is *not* set, the string is quoted and escaped.

If both in.is_string and in.dont_escape are set, the string is quoted but not escaped.

If in.is_string is not set, the data is inserted as-is. This is the case when you initialize **SQLString** (p. 140) with one of the constructors taking an integral type, for instance.

6.1.3.38 template<class Strng, class T> Strng stream2string (const T & *object*)

Converts a stream-able object to any type that can be initialized from an `std::string`.

This adapter takes any object that has an `out_stream()` member function and converts it to a string type. An example of such a type within the library is **mysqlpp::Date** (p. 65).

6.1.3.39 template<class Seq> value_list_b<Seq, do_nothing_type0> 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.

Same as value_list(Seq&, const char*, Manip, bool, bool...) but without the Manip or delimiter parameters. We use the do_nothing manipulator, meaning that the value list items are neither escaped nor quoted when being inserted into a stream. The delimiter is a comma. This form is suitable for lists of simple data, such as integers.

6.1.3.40 template<class Seq> value_list_b<Seq, do_nothing_type0> 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.

Same as value_list(Seq&, const char*, Manip, bool, bool...) but without the Manip parameter. We use the do_nothing manipulator, meaning that the value list items are neither escaped nor quoted when being inserted into a stream.

6.1.3.41 template<class Seq, class Manip> value_list_b<Seq, Manip> 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. 149) (sparse value list).

Same as value_list(Seq&, const char*, Manip, const vector<bool>&), except that it takes the bools as arguments instead of wrapped up in a vector object.

6.1.3.42 template<class Seq, class Manip> value_list_b<Seq, Manip> value_list (const Seq & *s*, const char * *d*, Manip *m*, const std::vector< bool > & *vb*) [inline]

Constructs a **value_list_b** (p. 149) (sparse value list).

Parameters:

s an STL sequence of items in the value list

d delimiter operator<< should place between items

m manipulator to use when inserting items into a stream

vb for each item in this vector that is true, the corresponding item in the value list is inserted into a stream; the others are suppressed

6.1.3.43 template<class Seq, class Manip> value_list_ba<Seq, Manip> value_list (const Seq & *s*, const char * *d*, Manip *m*)

Constructs a **value_list_ba** (p. 151).

Parameters:

s an STL sequence of items in the value list

d delimiter operator<< should place between items

m manipulator to use when inserting items into a stream

6.1.3.44 template<class Seq> value_list_ba<Seq, do_nothing_type0> value_list (const Seq & *s*, const char * *d* = ",")

Constructs a **value_list_ba** (p. 151).

This function returns a value list that uses the 'do_nothing' manipulator. That is, the items are not quoted or escaped in any way. See value_list(Seq, const char*, Manip) if you need to specify a manipulator.

Parameters:

s an STL sequence of items in the value list

d delimiter operator<< should place between items

6.1.4 Variable Documentation

6.1.4.1 bool mysqlpp::dont_quote_auto = false

Set (p. 129) to true if you want to suppress automatic quoting.

Works only for ColData inserted into C++ streams.

Chapter 7

MySQL++ Class Documentation

7.1 mysqlpp::BadConversion Class Reference

Exception thrown when a bad type conversion is attempted.

`#include <exceptions.h>`

Public Methods

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

  Create exception object, building error string dynamically.

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

  Create exception object, given completed error string.

- **BadConversion** (const std::string &wt="")

  Create exception object, with error string only.

- **~BadConversion** () throw ()

  Destroy exception object.

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

  Returns the error message.

Public Attributes

- const char * **type_name**

  name of type we tried to convert to

- const std::string **data**

  string form of data we tried to convert

- size_t **retrieved**

  documentation needed!

- size_t **actual_size**

  documentation needed!

7.1.1 Detailed Description

Exception thrown when a bad type conversion is attempted.

7.1.2 Constructor & Destructor Documentation

7.1.2.1 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 ??

7.1.2.2 mysqlpp::BadConversion::BadConversion (const std::string & *wt*, const char * *tn*, const char * *d*, size_t *r*, size_t *a*) [inline]

Create exception object, given completed error string.

Parameters:

wt the "what" error string

tn type name we tried to convert to

d string form of data we tried to convert

r ??

a ??

7.1.2.3 mysqlpp::BadConversion::BadConversion (const std::string & *wt* = "") [inline]

Create exception object, with error string only.

Parameters:

wt the "what" error string

All other data members are initialize to default values

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

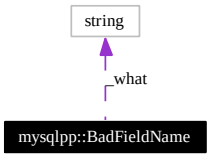
- **exceptions.h**

7.2 mysqlpp::BadFieldName Class Reference

Exception thrown when a requested named field doesn't exist.

`#include <exceptions.h>`

Collaboration diagram for mysqlpp::BadFieldName:



Public Methods

- **BadFieldName** (const char *bad_field)
  Create exception object.
- **~BadFieldName** () throw ()
  Destroy exception object.
- virtual const char * **what** () const throw ()
  Returns the error message.

7.2.1 Detailed Description

Exception thrown when a requested named field doesn't exist.

Thrown by **Row::lookup_by_name()** (p. 117) when you pass a field name that isn't in the result set.

7.2.2 Constructor & Destructor Documentation

7.2.2.1 mysqlpp::BadFieldName::BadFieldName (const char * *bad_field*) [inline]

Create exception object.

Parameters:
bad_field name of field the MySQL server didn't like

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

- **exceptions.h**

7.3 mysqlpp::BadNullConversion Class Reference

Exception thrown when you attempt to convert a SQL null to an incompatible type.

`#include <exceptions.h>`

Public Methods

- **BadNullConversion** (const std::string &wt="")

  Create exception object.

- **~BadNullConversion** () throw ()

  Destroy exception object.

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

  Returns the error message.

7.3.1 Detailed Description

Exception thrown when you attempt to convert a SQL null to an incompatible type.

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

- **exceptions.h**

7.4 mysqlpp::BadQuery Class Reference

Exception thrown when MySQL encounters a problem while processing your query.

`#include <exceptions.h>`

Public Methods

- **BadQuery** (const std::string &er="")

  Create exception object.

- **~BadQuery** () throw ()

  Destroy exception object.

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

  Returns the error message.

Public Attributes

- const std::string **error**

  explanation of why query was bad

7.4.1 Detailed Description

Exception thrown when MySQL encounters a problem while processing your query.

This is the most generic MySQL++ exception. It is thrown when your SQL syntax is incorrect, or a field you requested doesn't exist in the database, or....

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

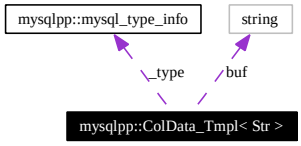
- **exceptions.h**

7.5 mysqlpp::ColData_Tmpl< Str > Class Template Reference

Template for string data that can convert itself to any standard C data type.

`#include <coldata.h>`

Collaboration diagram for mysqlpp::ColData_Tmpl< Str >:



Public Methods

- **ColData_Tmpl** ()

  Default constructor.

- **ColData_Tmpl** (bool n, **mysql_type_info** t=**mysql_type_info::string_type**)

  Constructor allowing you to set the null flag and the type data.

- **ColData_Tmpl** (const char *str, **mysql_type_info** t=**mysql_type_info::string_type**, bool n=false)

  Full constructor.

- **mysql_type_info type** () const

  Get this object's current MySQL type.

- bool **quote_q** () const

  Returns true if data of this type should be quoted, false otherwise.

- bool **escape_q** () const

  Returns true if data of this type should be escaped, false otherwise.

- template<class Type> Type **conv** (Type dummy) const

  Template for converting data from one type to another.

- void **it_is_null** ()

  Set (p. 129) *a flag indicating that this object is a SQL null.*

- const bool **is_null** () const

  Returns true if this object is a SQL null.

- const std::string & **get_string** () const

  Returns the string form of this object's data.

- **operator cchar** * () const

  Returns a const char pointer to the string form of this object's 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.

- template<class T, class B> **operator Null** () const

  Converts this object to a SQL null.

7.5.1 Detailed Description

template<class Str> class mysqlpp::ColData_Tmpl< Str >

Template for string data that can convert itself to any standard C data type.

Do not use this class directly. Use the typedef ColData or MutableColData instead. ColData is a `ColData_Tmpl<const std::string>` and MutableColData is a `ColData_Tmpl<std::string>`.

The ColData types add to the C++ string type the ability to automatically convert the string data to any of the basic C types. This is important with SQL, because all data coming from the database is in string form. MySQL++ uses this class internally to hold the data it receives from the server, so you can use it naturally, because it does the conversions implicitly:

```
ColData("12.86") + 2.0
```

That works fine, but be careful. If you had said this instead:

```
ColData("12.86") + 2
```

the result would be 14 because 2 is an integer, and C++'s type conversion rules put the ColData object in an integer context.

If these automatic conversions scare you, define the micro NO_BINARY_OPERS to disable this behavior.

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.

7.5.2 Constructor & Destructor Documentation

7.5.2.1 template<class Str> mysqlpp::ColData_Tmpl< Str >::ColData_Tmpl () [inline]

Default constructor.

Null (p. 94) flag is set to false, type data is not set, and string data is left empty.

It's probably a bad idea to use this ctor, becuase there's no way to set the type data once the object's constructed.

7.5.2.2 template<class Str> mysqlpp::ColData_Tmpl< Str >::ColData_Tmpl (bool *n*, mysql_type_info *t* = mysql_type_info::string_type) [inline, explicit]

Constructor allowing you to set the null flag and the type data.

Parameters:

n if true, data is a SQL null

t MySQL type information for data being stored

7.5.2.3 template<class Str> mysqlpp::ColData_Tmpl< Str >::ColData_Tmpl (const char * *str*, mysql_type_info *t* = mysql_type_info::string_type, bool *n* = false) [inline, explicit]

Full constructor.

Parameters:

str the string this object represents

t MySQL type information for data within str

n if true, str is a SQL null

7.5.3 Member Function Documentation

7.5.3.1 template<class Str> template<class T, class B> mysqlpp::ColData_Tmpl< Str >::operator Null< T, B > () const

Converts this object to a SQL null.

Returns null directly if the string data held by the object is exactly equal to "NULL". Else, it data.

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

- **coldata.h**

7.6 mysqlpp::Connection Class Reference

Manages the connection to the MySQL database.

`#include <connection.h>`

Public Methods

- **Connection** ()

  Create object without connecting it to the MySQL server.

- **Connection** (bool te)
- **Connection** (const char *db, const char *host="", const char *user="", const char *passwd="", bool te=true)

  For connecting to database without any special options.

- **Connection** (const char *db, const char *host, const char *user, const char *passwd, **uint** port, my_bool compress=0, unsigned int connect_timeout=60, bool te=true, **cchar** *socket_name=0, unsigned int client_flag=0)

  Connect to database, allowing you to specify all connection parameters.

- bool **connect** (**cchar** *db="", **cchar** *host="", **cchar** *user="", **cchar** *passwd="")

  Open connection to MySQL database.

- bool **real_connect** (**cchar** *db="", **cchar** *host="", **cchar** *user="", **cchar** *passwd="", **uint** port=0, my_bool compress=0, unsigned int connect_timeout=60, **cchar** *socket_name=0, unsigned int client_flag=0)

  Connect to database after object is created.

- void **close** ()

  Close connection to MySQL server.

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

  Calls MySQL C API function `mysql_info()` *and returns result as a C++ string.*

- bool **connected** () const

  return true if connection was established successfully

- bool **success** () const

  Return true if the last query was successful.

- bool **lock** ()

  Lock the object.

- void **unlock** ()

  Unlock the object.

- void **purge** ()

  Alias for **close()** (p. 46).

- **Query query** ()

  Return a new query object.

- **operator bool** ()

  Alias for **success()** (p. 42).

- 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.

- int **refresh** (unsigned int refresh_options)

  Wraps MySQL C API function `mysql_refresh()`.

- int **ping** ()

  "Pings" the MySQL database

- int **kill** (unsigned long pid)

  Kill a MySQL server thread.

- std::string **client_info** ()

  Get MySQL client library version.

- std::string **host_info** ()

  Get information about the network connection.

- int **proto_info** ()

  Returns version number of MySQL protocol this connection is using.

- std::string **server_info** ()

  Get the MySQL server's version number.

- std::string **stat** ()

  Returns information about MySQL server status.

- **Result store** (const std::string &str)

  Execute a query that can return a result set.

- **ResUse use** (const std::string &str)

  Execute a query that can return a result set.

- **ResNSel execute** (const std::string &str)

  Execute a query.

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

  Execute a query.

- **Result store** (const std::string &str, bool te)

  Same as store(str) except that it allows you to turn off exceptions for this query.

- **ResUse use** (const std::string &str, bool te)

  Same as use(str) except that it allows you to turn off exceptions for this query.

- **ResNSel execute** (const std::string &str, bool te)

  Same as execute(str) except that it allows you to turn off exceptions for this query.

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

  Create a database.

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

  Drop a database.

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

  Change to a different database.

- bool **select_db** (const char *db)

  Change to a different database.

- bool **reload** ()

  Ask MySQL server to reload the grant tables.

- bool **shutdown** ()

  Ask MySQL server to shut down.

- std::string **infoo** ()

  Alias for **info()** (p. 42).

- st_mysql_options **get_options** () const

  Return the connection options object.

- int **read_options** (enum mysql_option option, const char *arg)

  Sets the given MySQL connection option.

- my_ulonglong **affected_rows** ()

  Return the number of rows affected by the last query.

- my_ulonglong **insert_id** ()

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

- template<class Sequence> void **storein_sequence** (Sequence &con, const std::string &s)

  Execute a query, storing the result set in an STL sequence container.

- template<class Set> void **storein_set** (**Set** &con, const std::string &s)

  Execute a query, storing the result set in an STL associative container.

- template<class T> void **storein** (std::vector< T > &con, const std::string &s)

  Specialization of **storein_sequence()** (p. 52) *for* `std::vector`.

- template<class T> void **storein** (std::deque< T > &con, const std::string &s)

Specialization of **storein_sequence()** (p. 52) *for* `std::deque`.

- template<class T> void **storein** (std::list< T > &con, const std::string &s)
  Specialization of **storein_sequence()** (p. 52) *for* `std::list`.
- template<class T> void **storein** (std::set< T > &con, const std::string &s)
  Specialization of **storein_set()** (p. 52) *for* `std::set`.
- template<class T> void **storein** (std::multiset< T > &con, const std::string &s)
  Specialization of **storein_set()** (p. 52) *for* `std::multiset`.

7.6.1 Detailed Description

Manages the connection to the MySQL database.

7.6.2 Constructor & Destructor Documentation

7.6.2.1 mysqlpp::Connection::Connection ()

Create object without connecting it to the MySQL server.

Use **real_connect()** (p. 50) method to establish the connection.

7.6.2.2 mysqlpp::Connection::Connection (bool *te*)

Same as **default ctor** (p. 45) except that it allows you to choose whether exceptions are enabled.

Parameters:

te if true, exceptions are thrown on errors

7.6.2.3 mysqlpp::Connection::Connection (const char * *db*, const char * *host* = "", const char * *user* = "", const char * *passwd* = "", bool *te* = true)

For connecting to database without any special options.

This constructor takes the minimum parameters needed for most programs' use of MySQL. There is a `more complicated` constructor that lets you specify everything that the C API function `mysql_real_connect()` does.

Parameters:

db name of database to use

host host name or IP address of MySQL server, or 0 if server is running on the same host as your program

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

passwd password to use when logging in

te if true, throw exceptions on errors

7.6.2.4 mysqlpp::Connection::Connection (const char * *db*, const char * *host*, const char * *user*, const char * *passwd*, uint *port*, my_bool *compress* = 0, unsigned int *connect_timeout* = 60, bool *te* = true, cchar * *socket_name* = 0, unsigned int *client_flag* = 0)

Connect to database, allowing you to specify all connection parameters.

This constructor allows you to most fully specify the options used when connecting to the MySQL database. It is the thinnest layer in MySQL++ over the MySQL C API function `mysql_real_connect()`.

Parameters:

db name of database to use

host host name or IP address of MySQL server, or 0 if server is running on the same host as your program

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

passwd password to use when logging in

port TCP port number MySQL server is listening on, or 0 to use default value

compress if true, compress data passing through connection, to save bandwidth at the expense of CPU time

connect_timeout max seconds to wait for server to respond to our connection attempt

te if true, throw exceptions on errors

socket_name Unix domain socket server is using, if connecting to MySQL server on the same host as this program running on, or 0 to use default name

client_flag special connection flags. See MySQL C API documentation for `mysql_real_connect()` for details.

7.6.3 Member Function Documentation

7.6.3.1 my_ulonglong mysqlpp::Connection::affected_rows () [inline]

Return the number of rows affected by the last query.

Simply wraps `mysql_affected_rows()` in the C API.

7.6.3.2 std::string mysqlpp::Connection::client_info () [inline]

Get MySQL client library version.

Simply wraps `mysql_get_client_info()` in the C API.

7.6.3.3 void mysqlpp::Connection::close () [inline]

Close connection to MySQL server.

Closes the connection to the MySQL server.

7.6.3.4 bool mysqlpp::Connection::connect (cchar * *db* = "", cchar * *host* = "", cchar * *user* = "", cchar * *passwd* = "")

Open connection to MySQL database.

Open connection to the MySQL server, using defaults for all but the most common parameters. It's better to use one of the connect-on-create constructors if you can.

See `this` for parameter documentation.

7.6.3.5 bool mysqlpp::Connection::connected () const [inline]

return true if connection was established successfully

Returns:
true if connection was established successfully

7.6.3.6 bool mysqlpp::Connection::create_db (std::string *db*) [inline]

Create a database.

Parameters:
db name of database to create

Returns:
false if database was created successfully. (Yes, false! This behavior will change in the next major version.)

7.6.3.7 bool mysqlpp::Connection::drop_db (std::string *db*) [inline]

Drop a database.

Parameters:
db name of database to destroy

Returns:
false if database was created successfully. (Yes, false! This behavior will change in the next major version.)

7.6.3.8 int mysqlpp::Connection::errnum () [inline]

Return last MySQL error number associated with this connection.

Simply wraps `mysql_errno()` in the C API.

7.6.3.9 const char* mysqlpp::Connection::error () [inline]

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

Simply wraps `mysql_error()` in the C API.

7.6.3.10 bool mysqlpp::Connection::exec (const std::string & *str*)

Execute a query.

Same as **execute()** (p. 48), 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_query()`.

Parameters:
str the query to execute

Returns:
true if query was executed successfully

See also:
execute() (p. 48), **store()** (p. 51), **storein()** (p. 44), and **use()** (p. 53)

7.6.3.11 ResNSel mysqlpp::Connection::execute (const std::string & *str*, bool *te*)

Same as execute(str) except that it allows you to turn off exceptions for this query.

Parameters:
str query to execute

te if true, no exceptions will be thrown on errors.

7.6.3.12 ResNSel mysqlpp::Connection::execute (const std::string & *str*) [inline]

Execute a query.

Use this function if you don't expect the server to return a result set. For instance, a DELETE query. The returned **ResNSel** (p. 107) object contains status information from the server, such as whether the query succeeded, and if so how many rows were affected.

Parameters:
str the query to execute

Returns:
ResNSel (p. 107) status information about the query

See also:
exec() (p. 48), **store()** (p. 51), **storein()** (p. 44), and **use()** (p. 53)

7.6.3.13 std::string mysqlpp::Connection::host_info () [inline]

Get information about the network connection.

String contains info about type of connection and the server hostname.

Simply wraps `mysql_get_host_info()` in the C API.

7.6.3.14 std::string mysqlpp::Connection::infoo () [inline]

Alias for **info()** (p. 42).

This probably shouldn't exist. It will be removed in the next major version of the library unless a good justification is found.

7.6.3.15 my_ulonglong mysqlpp::Connection::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.

7.6.3.16 int mysqlpp::Connection::kill (unsigned long *pid*) [inline]

Kill a MySQL server thread.

Parameters:

pid ID of thread to kill

Simply wraps `mysql_kill()` in the C API.

7.6.3.17 bool mysqlpp::Connection::lock () [inline]

Lock the object.

Apparently supposed to prevent multiple simultaneous connections since only connection-related functions change the lock status but it's never actually checked! See Wishlist for plan to fix this.

7.6.3.18 mysqlpp::Connection::operator bool () [inline]

Alias for **success()** (p. 42).

Alias for **success()** (p. 42) member function. Allows you to have code constructs like this:

```
Connection conn;
.... use conn
if (conn) {
    ... last SQL query was successful
}
else {
    ... error occurred in SQL query
}
```

7.6.3.19 int mysqlpp::Connection::ping () [inline]

"Pings" the MySQL database

If server doesn't respond, this function tries to reconnect.

Return values:

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

nonzero if server did not respond to ping and we could not re-establish the connection

Simply wraps `mysql_ping()` in the C API.

7.6.3.20 int mysqlpp::Connection::proto_info () [inline]

Returns version number of MySQL protocol this connection is using.

Simply wraps `mysql_get_proto_info()` in the C API.

7.6.3.21 Query mysqlpp::Connection::query ()

Return a new query object.

The returned query object is tied to this MySQL connection, so when you call a method like **execute()** (p. 103) on that object, the query is sent to the server this object is connected to.

7.6.3.22 int mysqlpp::Connection::read_options (enum mysql_option *option*, const char * *arg*) [inline]

Sets the given MySQL connection option.

Not sure why this is named as it is; set_option() would be a better name. It may change in the next major version of the library.

Simply wraps `mysql_option()` in the C API.

7.6.3.23 bool mysqlpp::Connection::real_connect (cchar * *db* = "", cchar * *host* = "", cchar * *user* = "", cchar * *passwd* = "", uint *port* = 0, my_bool *compress* = 0, unsigned int *connect_timeout* = 60, cchar * *socket_name* = 0, unsigned int *client_flag* = 0)

Connect to database after object is created.

It's better to use one of the connect-on-create constructors if you can.

Despite the name, this function is not a direct wrapper for the MySQL C API function `mysql_real_connect()`. It also sets some connection-related options using `mysql_options()`.

See `this` for parameter documentation.

7.6.3.24 int mysqlpp::Connection::refresh (unsigned int *refresh_options*) [inline]

Wraps MySQL C API function `mysql_refresh()`.

The corresponding C API function is undocumented. All I know is that it's used by `mysqldump` and `mysqladmin`, according to MySQL bug database entry `http://bugs.mysql.com/bug.php?id=9816` If that entry changes to say that the function is now documented, reevaluate whether we need to wrap it. It may be that it's not supposed to be used by regular end-user programs.

7.6.3.25 bool mysqlpp::Connection::reload ()

Ask MySQL server to reload the grant tables.

User must have the "reload" privilege.

Simply wraps `mysql_reload()` in the C API. Since that function is deprecated, this one is, too. The MySQL++ replacement is execute("FLUSH PRIVILEGES").

7.6.3.26 std::string mysqlpp::Connection::server_info () [inline]

Get the MySQL server's version number.

Simply wraps `mysql_get_server_info()` in the C API.

7.6.3.27 bool mysqlpp::Connection::shutdown ()

Ask MySQL server to shut down.

User must have the "shutdown" privilege.

Simply wraps `mysql_shutdown()` in the C API.

7.6.3.28 std::string mysqlpp::Connection::stat () [inline]

Returns information about MySQL server status.

String 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.

7.6.3.29 Result mysqlpp::Connection::store (const std::string & *str*, bool *te*)

Same as store(str) except that it allows you to turn off exceptions for this query.

Parameters:

str query to execute

te if true, no exceptions will be thrown on errors.

7.6.3.30 Result mysqlpp::Connection::store (const std::string & *str*) [inline]

Execute a query that can return a result set.

This function returns the entire result set immediately. This is useful if you actually need all of the records at once, but if not, consider using **use()** (p. 53) instead, which returns the results one at a time. As a result of this difference, **use()** (p. 53) doesn't need to allocate as much memory as **store()** (p. 51).

You must use **store()** (p. 51), **storein()** (p. 44) or **use()** (p. 53) 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. 48) and **execute()** (p. 48) are more efficient.

The name of this method comes from the MySQL C API function it is implemented in terms of, `mysql_store_result()`.

Returns:
Result (p. 108) object containing entire result set

See also:
exec() (p. 48), **execute()** (p. 48), **storein()** (p. 44), and **use()** (p. 53)

7.6.3.31 template<class Sequence> void mysqlpp::Connection::storein_sequence (Sequence & *con*, const std::string & *s*)

Execute a query, storing the result set in an STL sequence container.

This function works much like **store()** (p. 51) from the caller's perspective, because it returns the entire result set at once. It's actually implemented in terms of **use()** (p. 53), however, so that memory for the result set doesn't need to be allocated twice.

Parameters:
con any STL sequence container, such as `std::vector`

s query string to execute

See also:
exec() (p. 48), **execute()** (p. 48), **store()** (p. 51), and **use()** (p. 53)

7.6.3.32 template<class Set> void mysqlpp::Connection::storein_set (Set & *con*, const std::string & *s*)

Execute a query, storing the result set in an STL associative container.

Parameters:
con any STL associative container, such as `std::set`

s query string to execute

The same thing as **storein_sequence()** (p. 52), 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.

7.6.3.33 void mysqlpp::Connection::unlock () [inline]

Unlock the object.

See **lock()** (p. 49) documentation for caveats.

7.6.3.34 ResUse mysqlpp::Connection::use (const std::string & *str*, bool *te*)

Same as use(str) except that it allows you to turn off exceptions for this query.

Parameters:
str query to execute

te if true, no exceptions will be thrown on errors.

7.6.3.35 ResUse mysqlpp::Connection::use (const std::string & *str*) [inline]

Execute a query that can return a result set.

Unlike **store()** (p. 51), this function does not return the result set directly. Instead, it returns an object that can walk through the result records one by one. This is superior to **store()** (p. 51) when there are a large number of results; **store()** (p. 51) 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.

Returns:
ResUse (p. 110) object that can walk through result set serially

See also:
exec() (p. 48), **execute()** (p. 48), **store()** (p. 51) and **storein()** (p. 44)

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

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

7.7 mysqlpp::const_string Class Reference

Wrapper for `const char*` to make it behave in a way more useful to MySQL++.

`#include <const_string.h>`

Public Types

- typedef const char **value_type**
  Type of the data stored in this object, when it is not equal to SQL null.
- typedef unsigned int **size_type**
  Type of "size" integers.
- typedef const char & **const_reference**
  Type used when returning a reference to a character in the string.
- typedef const char * **const_iterator**
  Type of iterators.
- typedef **const_iterator iterator**
  Same as const_iterator because the data cannot be changed.

Public Methods

- **const_string** ()
  Create empty string.
- **const_string** (const char *str)
  Initialize string from existing C string.
- const_string & **operator=** (const char *str)
  Assignment operator.
- **size_type size** () const
  Return number of characters in string.
- **const_iterator begin** () const
  Return iterator pointing to the first character of the string.
- **const_iterator end** () const
  Return iterator pointing to one past the last character of the string.
- **size_type length** () const
  Return number of characters in the string.
- **size_type max_size** () const
  Return the maximum number of characters in the string.

- **const_reference operator**[] (**size_type** pos) const

  Return a reference to a character within the string.

- **const_reference at** (**size_type** pos) const

  Return a reference to a character within the string.

- const char * **c_str** () const

  Return a const pointer to the string data, null-terminated.

- const char * **data** () const

  Alias for **c_str()** (p. 55).

- int **compare** (const const_string &str) const

  Lexically compare this string to another.

7.7.1 Detailed Description

Wrapper for `const char*` to make it behave in a way more useful to MySQL++.

This class implements a small subset of the standard string class.

Objects are created from an existing `const char*` variable by copying the pointer only. Therefore, the object pointed to by that pointer needs to exist for at least as long as the **const_string** (p. 54) object that wraps it.

7.7.2 Member Function Documentation

7.7.2.1 const_reference mysqlpp::const_string::at (size_type *pos*) const [inline]

Return a reference to a character within the string.

Unlike **operator**[]() (p. 55), this function throws an `std::out_of_range` exception if the index isn't within range.

7.7.2.2 int mysqlpp::const_string::compare (const const_string & *str*) const [inline]

Lexically compare this string to another.

Parameters:

str string to compare against this one

Return values:

<0 if str1 is lexically "less than" str2

0 if str1 is equal to str2

>0 if str1 is lexically "greater than" str2

7.7.2.3 size_type mysqlpp::const_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. 54); its size is always equal to the amount of data currently stored.

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

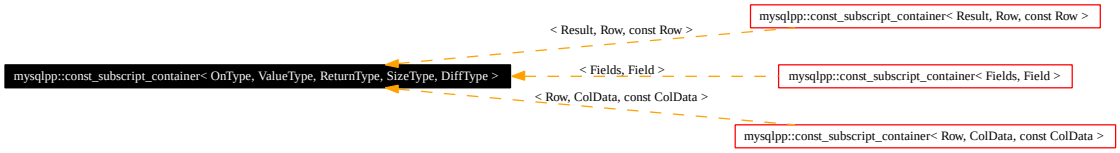
- **const_string.h**

7.8 mysqlpp::const_subscript_container< OnType, ValueType, ReturnType, SizeType, DiffType > Class Template Reference

A base class that one derives from to become a random access container, which can be accessed with subscript notation.

`#include <resiter.h>`

Inheritance diagram for mysqlpp::const_subscript_container< OnType, ValueType, ReturnType, SizeType, DiffType >:



Public Types

- typedef const_subscript_container< OnType, ValueType, ReturnType, SizeType, DiffType > **this_type**

  this object's type

- typedef **subscript_iterator**< const **this_type**, ReturnType, SizeType, DiffType > **iterator**

  mutable iterator type

- typedef **iterator const_iterator**

  constant iterator type

- typedef const std::reverse_iterator< **iterator** > **reverse_iterator**

  mutable reverse iterator type

- typedef const std::reverse_iterator< **const_iterator** > **const_reverse_iterator**

  const reverse iterator type

- typedef ValueType **value_type**

  type of data stored in container

- typedef **value_type** & **reference**

  reference to value_type

- typedef **value_type** & **const_reference**

  const ref to value_type

- typedef **value_type** * **pointer**

  pointer to value_type

- typedef **value_type** * **const_pointer**

const pointer to value_type

- typedef DiffType **difference_type**

  for index differences

- typedef SizeType **size_type**

  for returned sizes

Public Methods

- virtual **size_type size** () const=0

  Return count of elements in container.

- virtual ReturnType **operator**[] (SizeType i) const=0

  Return element at given index in container.

- **size_type max_size** () const

  Return maximum number of elements that can be stored in container without resizing.

- bool **empty** () const

  Returns true if container is empty.

- **iterator begin** () const

  Return iterator pointing to first element in the container.

- **iterator end** () const

  Return iterator pointing to one past the last element in the container.

- **reverse_iterator rbegin** () const

  Return reverse iterator pointing to first element in the container.

- **reverse_iterator rend** () const

  Return reverse iterator pointing to one past the last element in the container.

7.8.1 Detailed Description

template<class OnType, class ValueType, class ReturnType = const ValueType&, class SizeType = unsigned int, class DiffType = int> class mysqlpp::const_subscript_container< OnType, ValueType, ReturnType, SizeType, DiffType >

A base class that one derives from to become a random access container, which can be accessed with subscript notation.

OnType must have the member functions `operator[](SizeType)` and

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

- **resiter.h**

7.9 mysqlpp::cstr_equal_to Struct Reference

Function object that returns true if one const char* is equal to another.

`#include <compare.h>`

7.9.1 Detailed Description

Function object that returns true if one const char* is equal to another.

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

- **compare.h**

7.10 mysqlpp::cstr_greater Struct Reference

Function object that returns true if one const char* is lexically "greater than" another.

`#include <compare.h>`

7.10.1 Detailed Description

Function object that returns true if one const char* is lexically "greater than" another.

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

- **compare.h**

7.11 mysqlpp::cstr_greater_equal Struct Reference

Function object that returns true if one const char* is lexically "greater than or equal to" another.

`#include <compare.h>`

7.11.1 Detailed Description

Function object that returns true if one const char* is lexically "greater than or equal to" another.

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

- **compare.h**

7.12 mysqlpp::cstr_less Struct Reference

Function object that returns true if one const char* is lexically "less than" another.

`#include <compare.h>`

7.12.1 Detailed Description

Function object that returns true if one const char* is lexically "less than" another.

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

- **compare.h**

7.13 mysqlpp::cstr_less_equal Struct Reference

Function object that returns true if one const char* is lexically "less than or equal to" another.

`#include <compare.h>`

7.13.1 Detailed Description

Function object that returns true if one const char* is lexically "less than or equal to" another.

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

- **compare.h**

7.14 mysqlpp::cstr_not_equal_to Struct Reference

Function object that returns true if one const char* is not equal to another.

`#include <compare.h>`

7.14.1 Detailed Description

Function object that returns true if one const char* is not equal to another.

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

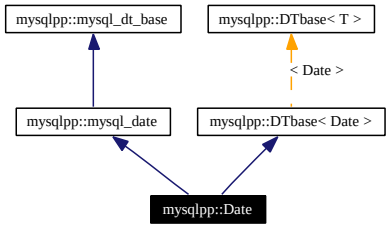
- **compare.h**

7.15 mysqlpp::Date Struct Reference

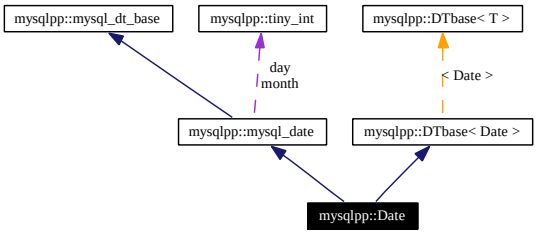
Holds MySQL dates.

`#include <datetime.h>`

Inheritance diagram for mysqlpp::Date:



Collaboration diagram for mysqlpp::Date:



Public Methods

- **Date** ()

  Default constructor.

- **Date** (**cchar** *str)

  Initialize object from a MySQL date string.

- **Date** (const **ColData** &str)

  Initialize object from a MySQL date string.

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

  Initialize object from a MySQL date string.

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

  Compare this date to another.

7.15.1 Detailed Description

Holds MySQL dates.

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

7.15.2 Constructor & Destructor Documentation

7.15.2.1 mysqlpp::Date::Date (cchar * *str*) [inline]

Initialize object from a MySQL date string.

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

7.15.2.2 mysqlpp::Date::Date (const ColData & *str*) [inline]

Initialize object from a MySQL date string.

See also:
Date(cchar*) (p. 66)

7.15.2.3 mysqlpp::Date::Date (const std::string & *str*) [inline]

Initialize object from a MySQL date string.

See also:
Date(cchar*) (p. 66)

7.15.3 Member Function Documentation

7.15.3.1 short int mysqlpp::Date::compare (const Date & *other*) const [inline, virtual]

Compare this date to another.

See also:
mysql_date::compare() (p. 81) for implementation

Implements **mysqlpp::DTbase< Date >** (p. 71).

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

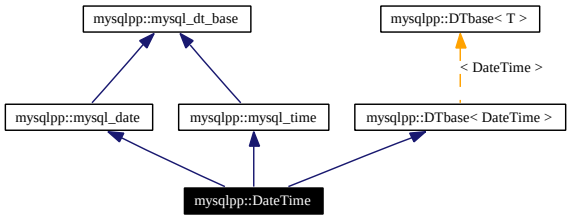
- **datetime.h**

7.16 mysqlpp::DateTime Struct Reference

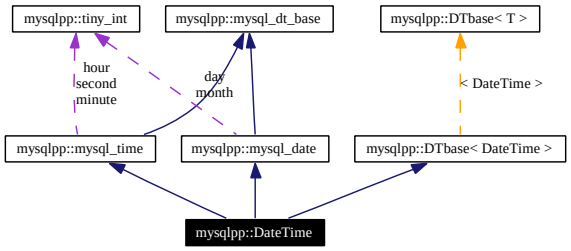
A combination of the **Date** (p. 65) and **Time** (p. 144) classes for holding MySQL DateTimes.

`#include <datetime.h>`

Inheritance diagram for mysqlpp::DateTime:



Collaboration diagram for mysqlpp::DateTime:



Public Methods

- **DateTime** ()

  Default constructor.

- **DateTime** (**cchar** *str)

  Initialize object from a MySQL date-and-time string.

- **DateTime** (const **ColData** &str)

  Initialize object from a MySQL date-and-time string.

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

  Initialize object from a MySQL date-and-time string.

- short int **compare** (const DateTime &other) const

  Compare this datetime to another.

- std::ostream & **out_stream** (std::ostream &os) const

  Insert the date and time into a stream.

- **cchar** * **convert** (**cchar** *)

  Parse a MySQL date and time string into this object.

7.16.1 Detailed Description

A combination of the **Date** (p. 65) and **Time** (p. 144) classes for holding MySQL DateTimes.

Objects of this class can be inserted into streams, and initialized from MySQL DATETIME strings.

7.16.2 Constructor & Destructor Documentation

7.16.2.1 mysqlpp::DateTime::DateTime (cchar * *str*) [inline]

Initialize object from a MySQL date-and-time string.

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

7.16.2.2 mysqlpp::DateTime::DateTime (const ColData & *str*) [inline]

Initialize object from a MySQL date-and-time string.

See also:
DateTime(cchar*) (p. 68)

7.16.2.3 mysqlpp::DateTime::DateTime (const std::string & *str*) [inline]

Initialize object from a MySQL date-and-time string.

See also:
DateTime(cchar*) (p. 68)

7.16.3 Member Function Documentation

7.16.3.1 short int mysqlpp::DateTime::compare (const DateTime & *other*) const [virtual]

Compare this datetime to another.

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

This method is protected because it is merely the engine used by the various operators in **DTbase** (p. 70).

Implements **mysqlpp::DTbase< DateTime >** (p. 71).

7.16.3.2 ostream & mysqlpp::DateTime::out_stream (std::ostream & *os*) const

Insert the date and time into a stream.

The date and time are inserted into the stream, in that order, with a space between them.

Parameters:
os stream to insert date and time into

Reimplemented from **mysqlpp::mysql_date** (p. 81).

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

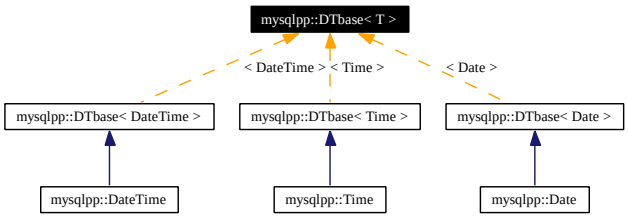
- **datetime.h**
- datetime.cpp

7.17 mysqlpp::DTbase< T > Struct Template Reference

Base class template for MySQL++ date and time classes.

`#include <datetime.h>`

Inheritance diagram for mysqlpp::DTbase< T >:



Public Methods

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

  Compare this object to another of the same type.

- 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.

7.17.1 Detailed Description

template<class T> struct mysqlpp::DTbase< T >

Base class template for MySQL++ date and time classes.

This template defines the comparison operators, which are all implemented in terms of **compare()** (p. 71). Each subclass implements that as a protected method, because these operators are the only supported comparison method.

7.17.2 Member Function Documentation

7.17.2.1 template<class T> virtual short int mysqlpp::DTbase< T >::compare (const T & *other*) const [pure virtual]

Compare this object to another of the same type.

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

Implemented in **mysqlpp::Date** (p. 66), **mysqlpp::Time** (p. 145), and **mysqlpp::DateTime** (p. 68).

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

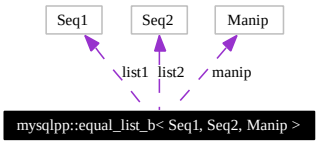
- **datetime.h**

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

Same as **equal_list_ba** (p. 74), 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 Methods

- **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 * **delem**

  delimiter to use between each pair of elements

- const char * **equl**

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

- Manip **manip**

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

7.18.1 Detailed Description

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

Same as **equal_list_ba** (p. 74), 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** (p. 74)'s documentation for more details.

7.18.2 Constructor & Destructor Documentation

7.18.2.1 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:

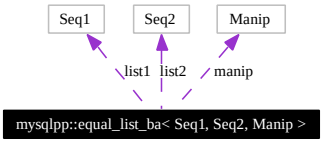
- **vallist.h**

7.19 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 Methods

- **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 * **delem**

  delimiter to use between each pair of elements

- const char * **equl**

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

- Manip **manip**

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

7.19.1 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()** (p. 24) functions build instances of this structure template to do their work. MySQL++'s SSQLS mechanism calls those functions when building SQL queries; you can call them yourself to do similar work. The "Harnessing SSQLS Internals" section of the user manual has some examples of this.

See also:
equal_list_b (p. 72)

7.19.2 Constructor & Destructor Documentation

7.19.2.1 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**

7.20 mysqlpp::FieldNames Class Reference

Holds a list of SQL field names.

`#include <field_names.h>`

Public Methods

- **FieldNames** ()

  Default constructor.

- **FieldNames** (const **ResUse** *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 **ResUse** *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.

- **uint operator**[ ] (std::string i) const

  Get the index number of a field given its name.

7.20.1 Detailed Description

Holds a list of SQL field names.

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

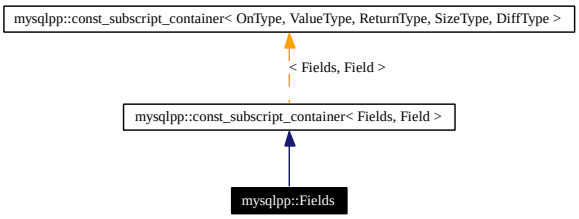
- **field_names.h**
- field_names.cpp

7.21 mysqlpp::Fields Class Reference

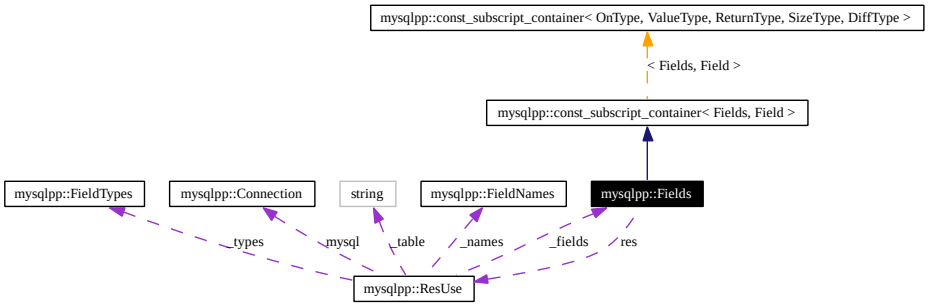
A container similar to `std::vector` for holding **mysqlpp::Field** (p. 14) records.

`#include <fields.h>`

Inheritance diagram for mysqlpp::Fields:



Collaboration diagram for mysqlpp::Fields:



Public Methods

- **Fields** ()

  Default constructor.

- **Fields** (**ResUse** *r)

  Create a field list from a result set.

- const **Field** & **operator**[] (**size_type** i) const

  Returns a field given its index.

- const **Field** & **operator**[] (int i) const

  Returns a field given its index.

- **size_type size** () const

  get the number of fields

7.21.1 Detailed Description

A container similar to `std::vector` for holding **mysqlpp::Field** (p. 14) records.

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

- **fields.h**
- fields.cpp

7.22 mysqlpp::FieldTypes Class Reference

A vector of SQL field types.

`#include <field_types.h>`

Public Methods

- **FieldTypes** ()
  Default constructor.
- **FieldTypes** (const **ResUse** *res)
  Create list of field types from a result set.
- **FieldTypes** (int i)
  Create fixed-size list of uninitialized field types.
- FieldTypes & **operator=** (const **ResUse** *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.
- **mysql_type_info** & **operator**[] (int i)
  Returns a field type within the list given its index.
- const **mysql_type_info** & **operator**[] (int i) const
  Returns a field type within the list given its index, in const context.

7.22.1 Detailed Description

A vector of SQL field types.

7.22.2 Member Function Documentation

7.22.2.1 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:

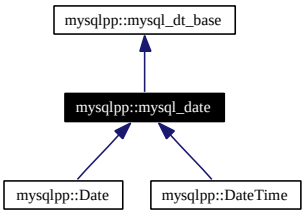
- **field_types.h**
- field_types.cpp

7.23 mysqlpp::mysql_date Struct Reference

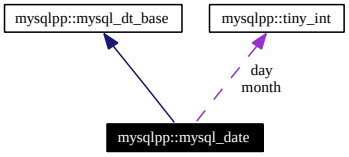
Base class of **Date** (p. 65).

`#include <datetime.h>`

Inheritance diagram for mysqlpp::mysql_date:



Collaboration diagram for mysqlpp::mysql_date:



Public Methods

- std::ostream & **out_stream** (std::ostream &os) const
  Insert the date into a stream.
- **cchar * convert** (**cchar** *)
  Parse a MySQL date string into this object.

Public Attributes

- short int **year**
  the year
- **tiny_int month**
  the month, 1-12
- **tiny_int day**
  the day, 1-31

Protected Methods

- short int **compare** (const mysql_date *other) const
  Compare this date to another.

7.23.1 Detailed Description

Base class of **Date** (p. 65).

7.23.2 Member Function Documentation

7.23.2.1 short int mysqlpp::mysql_date::compare (const mysql_date * *other*) const [protected]

Compare this date to another.

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

This method is protected because it is merely the engine used by the various operators in **DTbase** (p. 70).

7.23.2.2 ostream & mysqlpp::mysql_date::out_stream (std::ostream & *os*) const

Insert the date into a stream.

The format is YYYY-MM-DD, zero-padded.

Parameters:

os stream to insert date into

Reimplemented in **mysqlpp::DateTime** (p. 68).

7.23.3 Member Data Documentation

7.23.3.1 short int mysqlpp::mysql_date::year

the year

No surprises; the year 2005 is stored as the integer 2005.

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

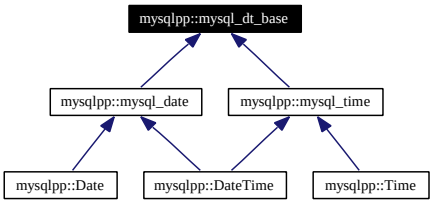
- **datetime.h**
- datetime.cpp

7.24 mysqlpp::mysql_dt_base Struct Reference

Base class for **mysql_date** (p. 80) and **mysql_time** (p. 83).

`#include <datetime.h>`

Inheritance diagram for mysqlpp::mysql_dt_base:



7.24.1 Detailed Description

Base class for **mysql_date** (p. 80) and **mysql_time** (p. 83).

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

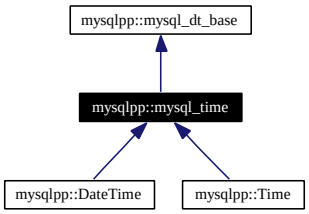
- **datetime.h**

7.25 mysqlpp::mysql_time Struct Reference

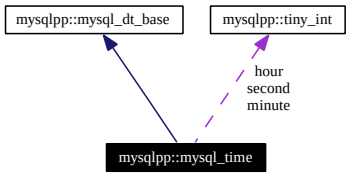
Base class of **Time** (p. 144).

`#include <datetime.h>`

Inheritance diagram for mysqlpp::mysql_time:



Collaboration diagram for mysqlpp::mysql_time:



Public Methods

- std::ostream & **out_stream** (std::ostream &os) const
  Insert the time into a stream.
- **cchar * convert** (**cchar** *)
  Parse a MySQL time string into this object.

Public Attributes

- **tiny_int hour**
  hour, 0-23
- **tiny_int minute**
  minute, 0-59
- **tiny_int second**
  second, 0-59

Protected Methods

- short int **compare** (const mysql_time *other) const
  Compare this time to another.

7.25.1 Detailed Description

Base class of **Time** (p. 144).

7.25.2 Member Function Documentation

7.25.2.1 short int mysqlpp::mysql_time::compare (const mysql_time * *other*) const [protected]

Compare this time to another.

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

This method is protected because it is merely the engine used by the various operators in **DTbase** (p. 70).

7.25.2.2 ostream & mysqlpp::mysql_time::out_stream (std::ostream & *os*) const

Insert the time into a stream.

The format is HH:MM:SS, zero-padded.

Parameters:

os stream to insert time into

Reimplemented in **mysqlpp::DateTime** (p. 68).

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

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

7.26 mysqlpp::mysql_type_info Class Reference

Holds basic type information for ColData.

`#include <type_info.h>`

Public Methods

- **mysql_type_info** ()
  Default constructor.
- **mysql_type_info** (unsigned char n)
  Create object.
- **mysql_type_info** (enum_field_types t, bool _unsigned, bool _null)
  Create object from MySQL C API type info.
- **mysql_type_info** (const MYSQL_FIELD &f)
  Create object from a MySQL C API field.
- **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=** (unsigned char n)
  Assign a new internal type value.
- mysql_type_info & **operator=** (const mysql_type_info &t)
  Assign another **mysql_type_info** (p. 85) *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 unsigned int **length** () const
  Return length of data in this field.
- const unsigned int **max_length** () const
  Return maximum length of data in this field.

- const mysql_type_info **base_type** () const

  Returns the type_info for the C++ type inside of the **mysqlpp::Null** (p. 94) *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.

Public Attributes

- unsigned int **_length**

  field length, from MYSQL_FIELD

- unsigned int **_max_length**

  max data length, from MYSQL_FIELD

Static Public Attributes

- const unsigned char **string_type** = 20

  The internal constant we use for our string type.

7.26.1 Detailed Description

Holds basic type information for ColData.

Class to hold basic type information for **mysqlpp::ColData** (p. 14).

7.26.2 Constructor & Destructor Documentation

7.26.2.1 mysqlpp::mysql_type_info::mysql_type_info (unsigned char *n*) [inline]

Create object.

Parameters:
n index into the internal type table

Because of the `n` parameter's definition, this constructor shouldn't be used outside the library.

7.26.2.2 mysqlpp::mysql_type_info::mysql_type_info (enum_field_types *t*, bool *_unsigned*, bool *_null*) [inline]

Create object from MySQL C API type info.

Parameters:

t the MySQL 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

7.26.2.3 mysqlpp::mysql_type_info::mysql_type_info (const MYSQL_FIELD & *f*) [inline]

Create object from a MySQL C API field.

Parameters:

f field from which we extract the type info

7.26.2.4 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.

7.26.3 Member Function Documentation

7.26.3.1 const mysql_type_info mysqlpp::mysql_type_info::base_type () [inline]

Returns the type_info for the C++ type inside of the **mysqlpp::Null** (p. 94) type.

Returns the type_info for the C++ type inside the **mysqlpp::Null** (p. 94) type. If the type is not **Null** (p. 94) then this is the same as **c_type()** (p. 87).

7.26.3.2 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.

7.26.3.3 const std::type_info & mysqlpp::mysql_type_info::c_type () [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.

7.26.3.4 bool mysqlpp::mysql_type_info::escape_q ()

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.

7.26.3.5 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.

7.26.3.6 const unsigned int mysqlpp::mysql_type_info::length () [inline]

Return length of data in this field.

This only works if you initialized this object from a MYSQL_FIELD object.

7.26.3.7 const unsigned int mysqlpp::mysql_type_info::max_length () [inline]

Return maximum length of data in this field.

This only works if you initialized this object from a MYSQL_FIELD object.

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

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

Returns the name that would be returned by typeid().**name()** (p. 88) for the C++ type associated with the SQL type.

7.26.3.9 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.

7.26.3.10 mysql_type_info& mysqlpp::mysql_type_info::operator= (unsigned char *n*) [inline]

Assign a new internal type value.

Parameters:
n an index into the internal MySQL++ type table

This function shouldn't be used outside the library.

7.26.3.11 bool mysqlpp::mysql_type_info::quote_q ()

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.

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

Returns the name of the SQL type.

Returns the SQL name for the type.

7.26.4 Member Data Documentation

7.26.4.1 const unsigned char mysqlpp::mysql_type_info::string_type = 20 [static]

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:

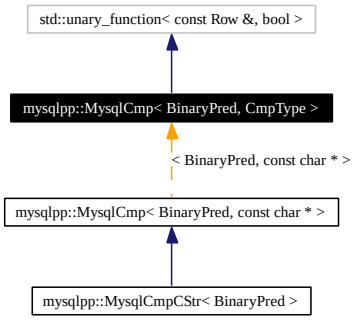
- **type_info.h**
- type_info.cpp

7.27 mysqlpp::MysqlCmp< BinaryPred, CmpType > Class Template Reference

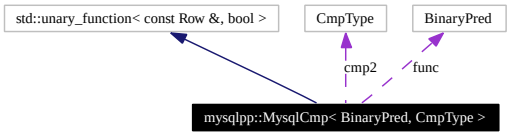
Template for making function objects that can compare something against a **Row** (p. 116) element.

`#include <compare.h>`

Inheritance diagram for mysqlpp::MysqlCmp< BinaryPred, CmpType >:



Collaboration diagram for mysqlpp::MysqlCmp< BinaryPred, CmpType >:



Public Methods

- **MysqlCmp** (**uint** i, const BinaryPred &f, const CmpType &c)

  MysqlCmp (p. 90) *constructor.*

- bool **operator()** (const **Row** &cmp1) const

  Run the predicate function on this row and the object's data, and return its value.

Protected Attributes

- unsigned int **index**

  Index of field within **Row** (p. 116) *object to compare against.*

- BinaryPred **func**

  Predicate function to use for the comparison.

- CmpType **cmp2**

  What to compare the **Row** (p. 116)*'s field against.*

7.27.1 Detailed Description

template<class BinaryPred, class CmpType> class mysqlpp::MysqlCmp< BinaryPred, CmpType >

Template for making function objects that can compare something against a **Row** (p. 116) element.

See also:
mysql_cmp (p. 24)

7.27.2 Constructor & Destructor Documentation

7.27.2.1 template<class BinaryPred, class CmpType> mysqlpp::MysqlCmp< BinaryPred, CmpType >::MysqlCmp (uint *i*, const BinaryPred & *f*, const CmpType & *c*) [inline]

MysqlCmp (p. 90) constructor.

Parameters:
i field number within a row to compare against
f predicate function
c what to compare row element against

operator() for this object compares **Row** (p. 116)[i] to c using f.

7.27.3 Member Function Documentation

7.27.3.1 template<class BinaryPred, class CmpType> bool mysqlpp::MysqlCmp< BinaryPred, CmpType >::operator() (const Row & *cmp1*) const [inline]

Run the predicate function on this row and the object's data, and return its value.

See the constructor's parameters for what we compare against.

Reimplemented in **mysqlpp::MysqlCmpCStr< BinaryPred >** (p. 93).

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

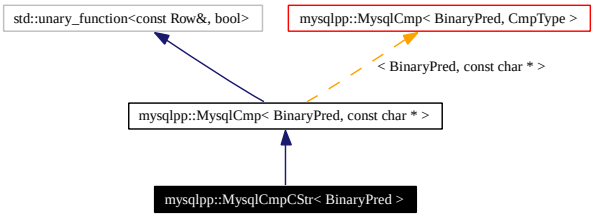
- **compare.h**

7.28 mysqlpp::MysqlCmpCStr< BinaryPred > Class Template Reference

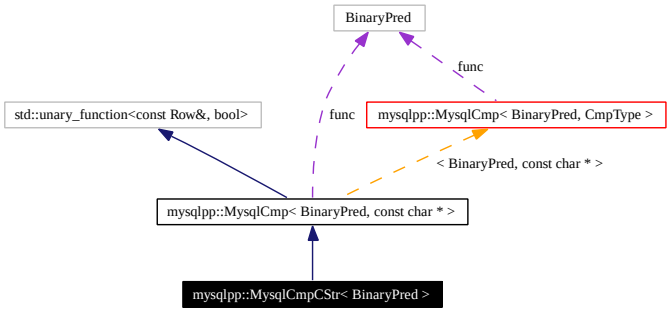
const char* specialization of **MysqlCmp** (p. 90)

`#include <compare.h>`

Inheritance diagram for mysqlpp::MysqlCmpCStr< BinaryPred >:



Collaboration diagram for mysqlpp::MysqlCmpCStr< BinaryPred >:



Public Methods

- **MysqlCmpCStr** (**uint** i, const BinaryPred &f, const char *c)

  MysqlCmpCStr (p. 92) *constructor.*

- bool **operator()** (const **Row** &cmp1) const

  Run the predicate function on this row and the object's data, and return its value.

7.28.1 Detailed Description

template<class BinaryPred> class mysqlpp::MysqlCmpCStr< BinaryPred >

const char* specialization of **MysqlCmp** (p. 90)

See also:
mysql_cmp_cstr (p. 25)

7.28.2 Constructor & Destructor Documentation

7.28.2.1 template<class BinaryPred> mysqlpp::MysqlCmpCStr< BinaryPred >::MysqlCmpCStr (uint *i*, const BinaryPred & *f*, const char * *c*) [inline]

MysqlCmpCStr (p. 92) constructor.

Parameters:

i field number within a row to compare against

f predicate function

c what to compare row element against

operator() for this object compares **Row** (p. 116)[i] to c using f.

7.28.3 Member Function Documentation

7.28.3.1 template<class BinaryPred> bool mysqlpp::MysqlCmpCStr< BinaryPred >::operator() (const Row & *cmp1*) const [inline]

Run the predicate function on this row and the object's data, and return its value.

See the constructor's parameters for what we compare against.

Reimplemented from **mysqlpp::MysqlCmp< BinaryPred, const char * >** (p. 91).

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

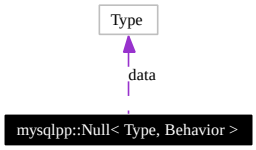
- **compare.h**

7.29 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 Methods

- **Null** ()
  Default constructor.
- **Null** (Type x)
  Initialize the object with a particular value.
- **Null** (const **null_type** &n)
  Construct a **Null** (p. 94) *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.

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.

7.29.1 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.

7.29.2 Constructor & Destructor Documentation

7.29.2.1 template<class Type, class Behavior = NullisNull> mysqlpp::Null< Type, Behavior >::Null () [inline]

Default constructor.

This ctor doesn't initialize any of the object's data members!

7.29.2.2 template<class Type, class Behavior = NullisNull> mysqlpp::Null< Type, Behavior >::Null (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.

7.29.2.3 template<class Type, class Behavior = NullisNull> mysqlpp::Null< Type, Behavior >::Null (const null_type & *n*) [inline]

Construct a **Null** (p. 94) equal to SQL null.

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

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

...to get a null `int`.

7.29.3 Member Function Documentation

7.29.3.1 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. 99), **NullisZero** (p. 100) and **NullisBlank** (p. 98).

Otherwise, just returns the 'data' member.

7.29.3.2 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.

7.29.3.3 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.

7.29.4 Member Data Documentation

7.29.4.1 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**

7.30 mysqlpp::null_type Class Reference

The type of the global **mysqlpp::null** (p. 21) object.

`#include <null.h>`

7.30.1 Detailed Description

The type of the global **mysqlpp::null** (p. 21) object.

This class is for internal use only. Normal code should use **Null** (p. 94) instead.

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

- **null.h**

7.31 mysqlpp::NullisBlank Struct Reference

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

`#include <null.h>`

7.31.1 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. 94)

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

- **null.h**

7.32 mysqlpp::NullisNull Struct Reference

Class for objects that define SQL null in terms of MySQL++'s **null_type** (p. 97).

`#include <null.h>`

7.32.1 Detailed Description

Class for objects that define SQL null in terms of MySQL++'s **null_type** (p. 97).

Returns a **null_type** (p. 97) 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. 94)

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

- **null.h**

7.33 mysqlpp::NullisZero Struct Reference

Class for objects that define SQL null as 0.

`#include <null.h>`

7.33.1 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. 94)

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

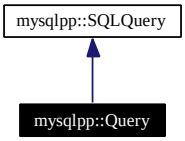
- **null.h**

7.34 mysqlpp::Query Class Reference

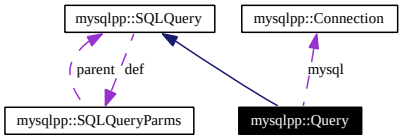
A class for building and executing SQL queries.

`#include <query.h>`

Inheritance diagram for mysqlpp::Query:



Collaboration diagram for mysqlpp::Query:



Public Methods

- **Query** (**Connection** *c, bool te=false)

  Create a new query object attached to a connection.

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

  Create a new query object as a copy of another.

- std::string **error** ()

  Get the last error message that happened on the connection we're bound to.

- bool **success** ()

  Returns true if the query executed successfully.

- std::string **preview** ()

  Return the query string currently in the buffer.

- std::string **preview** (**parms** &p)

  Return the query string currently in the buffer.

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

  Execute a query.

- **ResNSel execute** ()

  Execute a query.

- **ResUse use** ()

  Execute a query that can return a result set.

- **Result store** ()

  Execute a query that can return a result set.

- template<class Sequence> void **storein_sequence** (Sequence &con, **query_reset** r=RESET_QUERY)

  Execute a query, storing the result set in an STL sequence container.

- template<class Set> void **storein_set** (**Set** &con, **query_reset** r=RESET_QUERY)

  Execute a query, storing the result set in an STL associative container.

- template<class Container> void **storein** (Container &con, **query_reset** r=RESET_QUERY)

  Execute a query, and store the entire result set in an STL container.

- 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.

7.34.1 Detailed Description

A class for building and executing SQL queries.

This class is derived from **SQLQuery** (p. 132). It adds to that a tie between the query object and a MySQL++ **Connection** (p. 42) object, so that the query can be sent to the MySQL server we're connected to.

7.34.2 Constructor & Destructor Documentation

7.34.2.1 mysqlpp::Query::Query (Connection * *c*, bool *te* = false) [inline]

Create a new query object attached to a connection.

This is the constructor used by **mysqlpp::Connection::query()** (p. 50).

Parameters:

c connection the finished query should be sent out on

te if true, throw exceptions on errors

7.34.3 Member Function Documentation

7.34.3.1 bool mysqlpp::Query::exec (const std::string & *str*)

Execute a query.

Same as **execute()** (p. 103), 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_query()`.

Parameters:
str the query to execute

Returns:
true if query was executed successfully

See also:
execute() (p. 103), **store()** (p. 104), **storein()** (p. 105), and **use()** (p. 106)

7.34.3.2 ResNSel mysqlpp::Query::execute () [inline]

Execute a query.

Use this function if you don't expect the server to return a result set. For instance, a DELETE query. The returned **ResNSel** (p. 107) object contains status information from the server, such as whether the query succeeded, and if so how many rows were affected.

There are a number of overloaded versions of this function. The one without parameters simply executes a query that you have built up in the object in some way. (For instance, via the **insert()** (p. 104) method, or by using the object's stream interface.) You can also pass the function an std::string containing a SQL query, a **SQLQueryParms** (p. 138) object, or as many as 12 SQLStrings. The latter two (or is it 13?) overloads are for filling out template queries.

Returns:
ResNSel (p. 107) status information about the query

See also:
exec() (p. 103), **store()** (p. 104), **storein()** (p. 105), and **use()** (p. 106)

7.34.3.3 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. 104), **update()** (p. 105)

Reimplemented from **mysqlpp::SQLQuery** (p. 134).

7.34.3.4 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. 104), **update()** (p. 105)

Reimplemented from **mysqlpp::SQLQuery** (p. 135).

7.34.3.5 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. 104), **update()** (p. 105)

Reimplemented from **mysqlpp::SQLQuery** (p. 135).

7.34.3.6 Result mysqlpp::Query::store () [inline]

Execute a query that can return a result set.

This function returns the entire result set immediately. This is useful if you actually need all of the records at once, but if not, consider using **use()** (p. 106) instead, which returns the results one at a time. As a result of this difference, **use()** (p. 106) doesn't need to allocate as much memory as **store()** (p. 104).

You must use **store()** (p. 104), **storein()** (p. 105) or **use()** (p. 106) 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. 103) and **execute()** (p. 103) 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. 103).

Returns:
Result (p. 108) object containing entire result set

See also:
exec() (p. 103), **execute()** (p. 103), **storein()** (p. 105), and **use()** (p. 106)

7.34.3.7 template<class Container> void mysqlpp::Query::storein (Container & *con*, query_reset *r* = RESET_QUERY) [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. 105) or **storein_set()** (p. 105), 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. 103), **execute()** (p. 103), **store()** (p. 104), and **use()** (p. 106) for alternative query execution mechanisms.

7.34.3.8 template<class Sequence> void mysqlpp::Query::storein_sequence (Sequence & *con*, query_reset *r* = RESET_QUERY) [inline]

Execute a query, storing the result set in an STL sequence container.

This function works much like **store()** (p. 104) from the caller's perspective, because it returns the entire result set at once. It's actually implemented in terms of **use()** (p. 106), 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. 103), 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`

r whether the query automatically resets after being used

See also:

exec() (p. 103), **execute()** (p. 103), **store()** (p. 104), and **use()** (p. 106)

7.34.3.9 template<class Set> void mysqlpp::Query::storein_set (Set & *con*, query_reset *r* = RESET_QUERY) [inline]

Execute a query, storing the result set in an STL associative container.

The same thing as **storein_sequence()** (p. 105), 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.

7.34.3.10 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. 104), **replace()** (p. 104)

Reimplemented from **mysqlpp::SQLQuery** (p. 136).

7.34.3.11 ResUse mysqlpp::Query::use () [inline]

Execute a query that can return a result set.

Unlike **store()** (p. 104), this function does not return the result set directly. Instead, it returns an object that can walk through the result records one by one. This is superior to **store()** (p. 104) when there are a large number of results; **store()** (p. 104) 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. 103).

Returns:

ResUse (p. 110) object that can walk through result set serially

See also:

exec() (p. 103), **execute()** (p. 103), **store()** (p. 104) and **storein()** (p. 105)

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

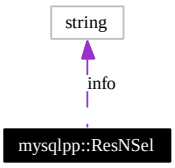
- **query.h**
- query.cpp

7.35 mysqlpp::ResNSel Class Reference

Holds the information on the success of queries that don't return any results.

`#include <result.h>`

Collaboration diagram for mysqlpp::ResNSel:



Public Methods

- **ResNSel** (**Connection** *q)
  Initialize object.
- **operator bool** ()
  Returns true if the query was successful.

Public Attributes

- bool **success**
  if true, query was successful
- my_ulonglong **insert_id**
  last value used for AUTO_INCREMENT field
- my_ulonglong **rows**
  number of rows affected
- std::string **info**
  additional info about query result

7.35.1 Detailed Description

Holds the information on the success of queries that don't return any results.

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

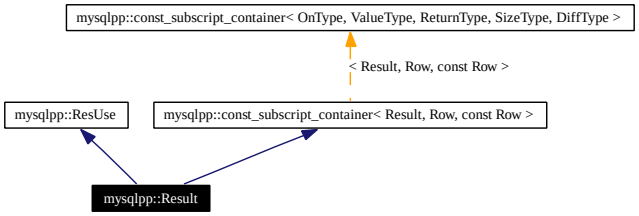
- **result.h**
- result.cpp

7.36 mysqlpp::Result Class Reference

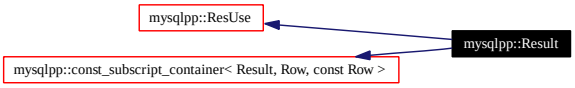
This class manages SQL result sets.

`#include <result.h>`

Inheritance diagram for mysqlpp::Result:



Collaboration diagram for mysqlpp::Result:



Public Methods

- **Result** ()
  Default constructor.
- **Result** (MYSQL_RES *result, bool te=false)
  Fully initialize object.
- **Result** (const Result &other)
  Initialize object as a copy of another **Result** (p. 108) *object.*
- const **Row fetch_row** () const
  Wraps mysql_fetch_row() in MySQL C API.
- my_ulonglong **num_rows** () const
  Wraps mysql_num_rows() in MySQL C API.
- void **data_seek** (**uint** offset) const
  Wraps mysql_data_seek() in MySQL C API.
- **size_type size** () const
  Alias for **num_rows()** (p. 108)*, only with different return type.*
- **size_type rows** () const
  Alias for **num_rows()** (p. 108)*, only with different return type.*
- const **Row operator**[] (**size_type** i) const
  Get the row with an offset of i.

7.36.1 Detailed Description

This class manages SQL result sets.

Objects of this class are created to manage the result of "store" queries, where the result set is handed to the program as single block of row data. (The name comes from the MySQL C API function `mysql_store_result()` which creates these blocks of row data.)

This class is a random access container (in the STL sense) which is neither less-than comparable nor assignable. This container provides a reverse random-access iterator in addition to the normal forward one.

7.36.2 Member Function Documentation

7.36.2.1 const Row mysqlpp::Result::fetch_row () const [inline]

Wraps mysql_fetch_row() in MySQL C API.

This is simply the const version of the same function in our **parent class** (p. 110) . Why this cannot actually *be* in our parent class is beyond me.

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

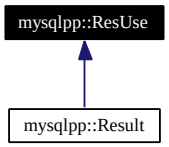
- **result.h**

7.37 mysqlpp::ResUse Class Reference

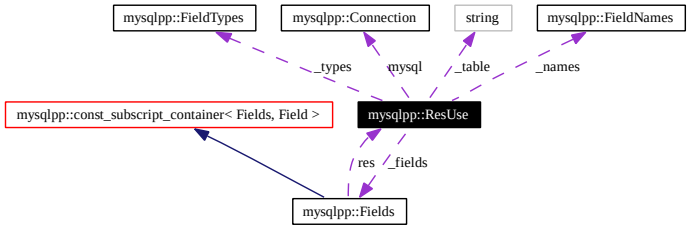
A basic result set class, for use with "use" queries.

`#include <result.h>`

Inheritance diagram for mysqlpp::ResUse:



Collaboration diagram for mysqlpp::ResUse:



Public Methods

- **ResUse** ()

  Default constructor.

- **ResUse** (MYSQL_RES *result, **Connection** *m=0, bool te=false)

  Create the object, fully initialized.

- **ResUse** (const ResUse &other)

  Create a copy of another **ResUse** (p. 110) *object.*

- **~ResUse** ()

  Destroy object.

- ResUse & **operator=** (const ResUse &other)

  Copy another **ResUse** (p. 110) *object's data into this object.*

- MYSQL_RES * **mysql_result** ()

  Return raw MySQL C API result set.

- **Row fetch_row** ()

  Wraps mysql_fetch_row() in MySQL C API.

- bool **eof** () const

  Wraps mysql_eof() in MySQL C API.

- unsigned long * **fetch_lengths** () const
  Wraps mysql_fetch_lengths() in MySQL C API.
- **Field** & **fetch_field** () const
  Wraps mysql_fetch_field() in MySQL C API.
- void **field_seek** (int field)
  Wraps mysql_field_seek() in MySQL C API.
- int **num_fields** () const
  Wraps mysql_num_fields() in MySQL C API.
- void **parent_leaving** ()
  Documentation needed!
- void **purge** ()
  Free all resources held by the object.
- **operator bool** () const
  Return true if we have a valid result set.
- unsigned int **columns** () const
  Return the number of columns in the result set.
- std::string & **table** ()
  Get the name of table that the result set comes from.
- const std::string & **table** () const
  Return the name of the table.
- int **field_num** (const std::string &) const
  Get the index of the named field.
- std::string & **field_name** (int)
  Get the name of the field at the given index.
- const std::string & **field_name** (int) const
  Get the name of the field at the given index.
- **FieldNames** & **field_names** ()
  Get the names of the fields within this result set.
- const **FieldNames** & **field_names** () const
  Get the names of the fields within this result set.
- void **reset_field_names** ()
  Reset the names in the field list to their original values.
- **mysql_type_info** & **field_type** (int i)
  Get the MySQL type for a field given its index.

- const **mysql_type_info** & **field_type** (int) const
  Get the MySQL type for a field given its index.
- **FieldTypes** & **field_types** ()
  Get a list of the types of the fields within this result set.
- const **FieldTypes** & **field_types** () const
  Get a list of the types of the fields within this result set.
- void **reset_field_types** ()
  Reset the field types to their original values.
- int **names** (const std::string &s) const
  Alias for **field_num()** (p. 114).
- std::string & **names** (int i)
  Alias for **field_name()** (p. 114).
- const std::string & **names** (int i) const
  Alias for **field_name()** (p. 114).
- **FieldNames** & **names** ()
  Alias for **field_names()** (p. 111).
- const **FieldNames** & **names** () const
  Alias for **field_names()** (p. 111).
- void **reset_names** ()
  Alias for **reset_field_names()** (p. 111).
- **mysql_type_info** & **types** (int i)
  Alias for **field_type()** (p. 111).
- const **mysql_type_info** & **types** (int i) const
  Alias for **field_type()** (p. 111).
- **FieldTypes** & **types** ()
  Alias for **field_types()** (p. 112).
- const **FieldTypes** & **types** () const
  Alias for **field_types()** (p. 112).
- void **reset_types** ()
  Alias for **reset_field_types()** (p. 112).
- const **Fields** & **fields** () const
  Get the underlying **Fields** (p. 77) *structure.*
- const **Field** & **fields** (unsigned int i) const

Get the underlying Field structure given its index.

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

  Returns true if the other **ResUse** (p. 110) *object shares the same underlying C API result set as this one.*

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

  Returns true if the other **ResUse** (p. 110) *object has a different underlying C API result set from this one.*

Protected Methods

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

  copy another **ResUse** (p. 110) *object's contents into this one.*

Protected Attributes

- **Connection** * **mysql**

  server result set comes from

- MYSQL_RES * **mysql_res**

  underlying C API result set

- bool **throw_exceptions**

  if true, throw exceptions on errors

- bool **initialized**

  if true, object is fully initted

- **FieldNames** * **_names**

  list of field names in result

- **FieldTypes** * **_types**

  list of field types in result

- **Fields _fields**

  list of fields in result

- std::string **_table**

  table result set comes from

7.37.1 Detailed Description

A basic result set class, for use with "use" queries.

A "use" query is one where you make the query and then process just one row at a time in the result instead of dealing with them all as a single large chunk. (The name comes from the MySQL C API function that initiates this action, `mysql_use_result().`) By calling **fetch_row()**

(p. 114) until it throws a **mysqlpp::BadQuery** (p. 37) exception (or an empty row if exceptions are disabled), you can process the result set one row at a time.

7.37.2 Member Function Documentation

7.37.2.1 void mysqlpp::ResUse::copy (const ResUse & *other*) [protected]

copy another **ResUse** (p. 110) object's contents into this one.

Not to be used on the self. Self-copy is not allowed.

7.37.2.2 Row mysqlpp::ResUse::fetch_row () [inline]

Wraps mysql_fetch_row() in MySQL C API.

This is not a thin wrapper. It does a lot of error checking before returning the **mysqlpp::Row** (p. 116) object containing the row data.

7.37.2.3 std::string & mysqlpp::ResUse::field_name (int) [inline]

Get the name of the field at the given index.

This is the inverse of **field_num()** (p. 114).

7.37.2.4 int mysqlpp::ResUse::field_num (const std::string &) const [inline]

Get the index of the named field.

This is the inverse of **field_name()** (p. 114).

7.37.2.5 mysqlpp::ResUse::operator bool () const [inline]

Return true if we have a valid result set.

This operator is primarily used to determine if a query was successful:

```
Query q("....");
if (q.use()) {
    ...
```

Query::use() (p. 106) returns a **ResUse** (p. 110) object, and it won't contain a valid result set if the query failed.

7.37.2.6 bool mysqlpp::ResUse::operator== (const ResUse & *other*) const [inline]

Returns true if the other **ResUse** (p. 110) object shares the same underlying C API result set as this one.

This works because the underlying result set is stored as a pointer, and thus can be copied and then compared.

7.37.2.7 void mysqlpp::ResUse::purge () [inline]

Free all resources held by the object.

This class's destructor is little more than a call to **purge()** (p. 115), so you can think of this as a way to re-use a **ResUse** (p. 110) object, to avoid having to completely re-create it.

7.37.2.8 const std::string& mysqlpp::ResUse::table () const [inline]

Return the name of the table.

This is only valid

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

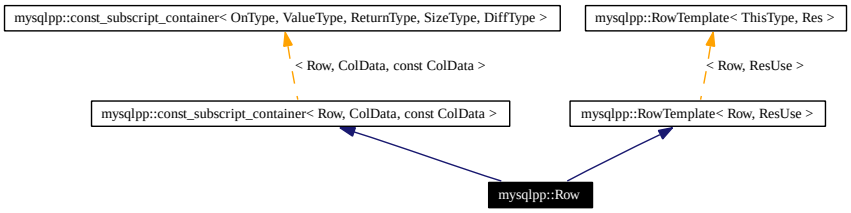
- **result.h**
- result.cpp

7.38 mysqlpp::Row Class Reference

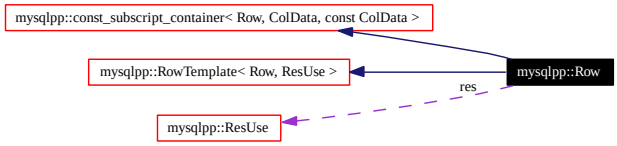
Manages rows from a result set.

`#include <row.h>`

Inheritance diagram for mysqlpp::Row:



Collaboration diagram for mysqlpp::Row:



Public Methods

- **Row** ()

  Default constructor.

- **Row** (MYSQL_ROW d, const **ResUse** *r, unsigned long *jj, bool te=false)

  Create a row object.

- **~Row** ()

  Destroy object.

- const Row & **self** () const

  Get a const reference to this object.

- Row & **self** ()

  Get a reference to this object.

- const **ResUse** & **parent** () const

  Get a reference to our parent class.

- **size_type size** () const

  Get the number of fields in the row.

- const **ColData operator**[ ] (**size_type** i) const

  Get the value of a field given its index.

- const **ColData lookup_by_name** (const char *) const

  Get the value of a field given its field name.

- const char * **raw_data** (int i) const

  Return the value of a field given its index, in raw form.

- **operator bool** () const

  Returns true if there is data in the row.

7.38.1 Detailed Description

Manages rows from a result set.

7.38.2 Constructor & Destructor Documentation

7.38.2.1 mysqlpp::Row::Row (MYSQL_ROW *d*, const ResUse * *r*, unsigned long * *jj*, bool *te* = false) [inline]

Create a row object.

Parameters:

d MySQL C API row data

r result set that the row comes from

jj length of each item in d

te if true, throw exceptions on errors

7.38.3 Member Function Documentation

7.38.3.1 const ColData mysqlpp::Row::lookup_by_name (const char *) const

Get the value of a field given its field name.

This function is rather inefficient. You should use operator[] if you're using Rows directly, or SSQLS for efficient named access to row elements.

7.38.3.2 const ColData mysqlpp::Row::operator[] (size_type *i*) const

Get the value of a field given its index.

If the array index is out of bounds, the C++ standard says that the underlying vector container should throw an exception. Whether it actually does is probably implementation-dependent.

Note that we return the **ColData** (p. 38) object by value. The purpose of ColData is to make it easy to convert the string data returned by the MySQL server to some more appropriate type, so you're almost certain to use this operator in a construct like this:

```
string s = row[2];
```

That accesses the third field in the row, returns a temporary ColData object, which is then automatically converted to a `std::string` and copied into `s`. That works fine, but beware of this similar but incorrect construct:

```
const char* pc = row[2];
```

This one line of code does what you expect, but `pc` is then a dangling pointer: it points to memory owned by the temporary ColData object, which will have been destroyed by the time you get around to actually *using* the pointer.

7.38.3.3 const char* mysqlpp::Row::raw_data (int *i*) const [inline]

Return the value of a field given its index, in raw form.

This is the same thing as operator[], except that the data isn't converted to a ColData object first. Also, this method does not check for out-of-bounds array indices.

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

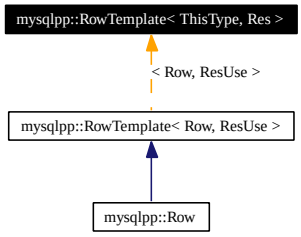
- **row.h**
- row.cpp

7.39 mysqlpp::RowTemplate< ThisType, Res > Class Template Reference

Base class for class **Row** (p. 116).

`#include <row.h>`

Inheritance diagram for mysqlpp::RowTemplate< ThisType, Res >:



Public Methods

- virtual **~RowTemplate** ()

  Destroy object.

- virtual const Res & **parent** () const=0

  Get a reference to our "parent" class.

- **value_list_ba**< ThisType, **quote_type0** > **value_list** (const char *d=",") const

  Get a list of the values in this row.

- template<class Manip> **value_list_ba**< ThisType, Manip > **value_list** (const char *d, Manip m) const

  Get a list of the values in this row.

- template<class Manip> **value_list_b**< ThisType, Manip > **value_list** (const char *d, Manip m, const std::vector< bool > &vb) const

  Get a list of the values in this row.

- **value_list_b**< ThisType, **quote_type0** > **value_list** (const char *d, const std::vector< bool > &vb) const

  Get a list of the values in this row.

- **value_list_b**< ThisType, **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**< ThisType, 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**< ThisType, **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**< ThisType, **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**< ThisType, 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**< ThisType, **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**< ThisType, **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.

- **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.

- template<class Manip> **value_list_b< FieldNames**, Manip > **field_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 field names in this row.

- **value_list_b< FieldNames, quote_type0 > field_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 field names in this row.

- **value_list_b< FieldNames, quote_type0 > field_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 field names in this row.

- **equal_list_ba< FieldNames**, ThisType, **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**, ThisType, Manip > **equal_list** (const char *d, const char *e, Manip m) const

  Get an "equal list" of the fields and values in this row.

Protected Methods

- virtual ThisType & **self** ()=0

  Return a pointer to this object.

- virtual const ThisType & **self** () const=0

  Return a const pointer to this object, for calls in const context.

7.39.1 Detailed Description

template<class ThisType, class Res> class mysqlpp::RowTemplate< ThisType, Res >

Base class for class **Row** (p. 116).

7.39.2 Member Function Documentation

7.39.2.1 template<class ThisType, class Res> template<class Manip> equal_list_ba<FieldNames, ThisType, Manip> mysqlpp::RowTemplate< ThisType, Res >::equal_list (const char * *d*, const char * *e*, Manip *m*) const [inline]

Get an "equal list" of the fields and values in this row.

This method's parameters govern how the returned list will behave when you insert it into a C++ stream:

Parameters:

d delimiter to use between items

e the operator to use between elements

m the manipulator to use for each element

For example, if d is ",", e is " = ", and m is the quote manipulator, then the field and value lists (a, b) (c, d'e) will yield an equal list that gives the following when inserted into a C++ stream:

```
'a' = 'c', 'b' = 'd''e'
```

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

7.39.2.2 template<class ThisType, class Res> equal_list_ba<FieldNames, ThisType, quote_type0> mysqlpp::RowTemplate< ThisType, Res >::equal_list (const char * *d* = ",", const char * *e* = " = ") const [inline]

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.

7.39.2.3 template<class ThisType, class Res> value_list_b<FieldNames, quote_type0> mysqlpp::RowTemplate< ThisType, Res >::field_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 field names in this row.

The 's' parameters name the field names that will be added to the returned list. When inserted into a C++ stream, a comma will be used as a delimiter between the items, and the items will be quoted and escaped.

7.39.2.4 template<class ThisType, class Res> value_list_b<FieldNames, quote_type0> mysqlpp::RowTemplate< ThisType, Res >::field_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 field names in this row.

The 's' parameters name the field names 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 items will be quoted and escaped.

7.39.2.5 template<class ThisType, class Res> template<class Manip> value_list_b<FieldNames, Manip> mysqlpp::RowTemplate< ThisType, Res >::field_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 field names in this row.

The 's' parameters name the field names 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.

7.39.2.6 template<class ThisType, class Res> value_list_b<FieldNames, quote_type0> mysqlpp::RowTemplate< ThisType, Res >::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 [inline]

Get a list of the field names in this row.

For each true parameter, the field name in that position within the row is added to the returned list. When the list is inserted into a C++ stream, a comma will be placed between the items as a delimiter, and the items will be quoted and escaped.

7.39.2.7 template<class ThisType, class Res> value_list_b<FieldNames, quote_type0> mysqlpp::RowTemplate< ThisType, Res >::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 [inline]

Get a list of the field names in this row.

For each true parameter, the field name in that position within the row is added to the returned list. When the list is inserted into a C++ stream, the delimiter 'd' will be placed between the items as a delimiter, and the items will be quoted and escaped.

7.39.2.8 template<class ThisType, class Res> template<class Manip> value_list_b<FieldNames, Manip> mysqlpp::RowTemplate< ThisType, Res >::field_list (const char * *d*, Manip *m*, bool *t0*, bool *t1* = false, bool *t2* = false, bool *t3* = false, bool *t4* = false, bool *t5* = false, bool *t6* = false, bool *t7* = false, bool *t8* = false, bool *t9* = false, bool *ta* = false, bool *tb* = false, bool *tc* = false) const [inline]

Get a list of the field names in this row.

For each true parameter, the field name in that position within the row is added to the returned list. When the list is inserted into a C++ stream, the delimiter 'd' will be placed between the items as a delimiter, and the manipulator 'm' used before each item.

7.39.2.9 template<class ThisType, class Res> value_list_b<FieldNames, quote_type0> mysqlpp::RowTemplate< ThisType, Res >::field_list (const std::vector< bool > & *vb*) const [inline]

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

7.39.2.10 template<class ThisType, class Res> value_list_b<FieldNames, quote_type0> mysqlpp::RowTemplate< ThisType, Res >::field_list (const char * *d*, const std::vector< bool > & *vb*) const [inline]

Get a list of the field names in this row.

Parameters:

d delimiter to place between the items when the list is inserted into a C++ stream

vb for each true item in this list, add that field name to the returned list; ignore the others

Field names will be quoted and escaped when inserted into a C++ stream.

7.39.2.11 template<class ThisType, class Res> template<class Manip> value_list_b<FieldNames, Manip> mysqlpp::RowTemplate< ThisType, Res >::field_list (const char * *d*, Manip *m*, const std::vector< bool > & *vb*) const [inline]

Get a list of the field names in this row.

Parameters:

d delimiter to place between the items when the list is inserted into a C++ stream

m manipulator to use before each item when the list is inserted into a C++ stream

vb for each true item in this list, add that field name to the returned list; ignore the others

7.39.2.12 template<class ThisType, class Res> template<class Manip> value_list_ba<FieldNames, Manip> mysqlpp::RowTemplate< ThisType, Res >::field_list (const char * *d*, Manip *m*) const [inline]

Get a list of the field names in this row.

Parameters:

d delimiter to place between the items when the list is inserted into a C++ stream

m manipulator to use before each item when the list is inserted into a C++ stream

7.39.2.13 template<class ThisType, class Res> value_list_ba<FieldNames, do_nothing_type0> mysqlpp::RowTemplate< ThisType, Res >::field_list (const char * *d* = ",") const [inline]

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.

7.39.2.14 template<class ThisType, class Res> virtual const Res& mysqlpp::RowTemplate< ThisType, Res >::parent () [pure virtual]

Get a reference to our "parent" class.

The meaning of this function is up to whatever class derives from this one.

Implemented in **mysqlpp::Row** (p. 116).

7.39.2.15 template<class ThisType, class Res> virtual ThisType& mysqlpp::RowTemplate< ThisType, Res >::self () [protected, pure virtual]

Return a pointer to this object.

Not sure what value this has over the 'this' pointer, but...

Implemented in **mysqlpp::Row** (p. 116).

7.39.2.16 template<class ThisType, class Res> value_list_b<ThisType, quote_type0> mysqlpp::RowTemplate< ThisType, Res >::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.

7.39.2.17 template<class ThisType, class Res> value_list_b<ThisType, quote_type0> mysqlpp::RowTemplate< ThisType, Res >::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.

7.39.2.18 template<class ThisType, class Res> template<class Manip> value_list_b<ThisType, Manip> mysqlpp::RowTemplate< ThisType, Res >::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.

7.39.2.19 template<class ThisType, class Res> value_list_b<ThisType, quote_type0> mysqlpp::RowTemplate< ThisType, Res >::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.

7.39.2.20 template<class ThisType, class Res> value_list_b<ThisType, quote_type0> mysqlpp::RowTemplate< ThisType, Res >::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.

7.39.2.21 template<class ThisType, class Res> template<class Manip> value_list_b<ThisType, Manip> mysqlpp::RowTemplate< ThisType, Res >::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.

7.39.2.22 template<class ThisType, class Res> value_list_b<ThisType, quote_type0> mysqlpp::RowTemplate< ThisType, Res >::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.

7.39.2.23 template<class ThisType, class Res> value_list_b<ThisType, quote_type0> mysqlpp::RowTemplate< ThisType, Res >::value_list (const char * *d*, const std::vector< bool > & *vb*) 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

Items will be quoted and escaped when inserted into a C++ stream.

7.39.2.24 template<class ThisType, class Res> template<class Manip> value_list_b<ThisType, Manip> mysqlpp::RowTemplate< ThisType, Res >::value_list (const char * *d*, Manip *m*, const std::vector< bool > & *vb*) const [inline]

Get a list of the values in this row.

Parameters:

d delimiter to use between values

m manipulator to use when inserting values into a stream

vb for each true item in this list, add that value to the returned list; ignore the others

7.39.2.25 template<class ThisType, class Res> template<class Manip> value_list_ba<ThisType, Manip> mysqlpp::RowTemplate< ThisType, Res >::value_list (const char * *d*, Manip *m*) const [inline]

Get a list of the values in this row.

Parameters:

d delimiter to use between values

m manipulator to use when inserting values into a stream

7.39.2.26 template<class ThisType, class Res> value_list_ba<ThisType, quote_type0> mysqlpp::RowTemplate< ThisType, Res >::value_list (const char * *d* = ",") 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 items will be quoted and escaped.

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

- **row.h**

7.40 mysqlpp::Set< Container > Class Template Reference

A special std::set derivative for holding MySQL data sets.

`#include <myset.h>`

Public Methods

- **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 **ColData** &str)

  Create object from a comma-separated list of values.

- std::ostream & **out_stream** (std::ostream &s) const

  Insert this set's data into a C++ stream in comma-separated format.

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

  Convert this set's data to a string containing comma-separated items.

7.40.1 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:

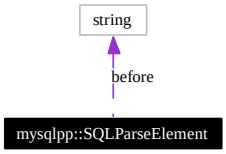
- **myset.h**

7.41 mysqlpp::SQLParseElement Struct Reference

Used within **SQLQuery** (p. 132) to hold elements for parameterized queries.

`#include <sql_query.h>`

Collaboration diagram for mysqlpp::SQLParseElement:



Public Methods

- **SQLParseElement** (std::string b, char o, char n)

  Create object.

Public Attributes

- std::string **before**

  string inserted before the parameter

- char **option**

  the parameter option, or blank if none

- char **num**

  the parameter position to use

7.41.1 Detailed Description

Used within **SQLQuery** (p. 132) 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. 130) 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. 101) method call. An example of how this can be helpful is in the "Template Queries" chapter of the user manual.

7.41.2 Constructor & Destructor Documentation

7.41.2.1 mysqlpp::SQLParseElement::SQLParseElement (std::string *b*, char *o*, 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:

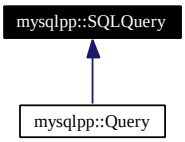
- **sql_query.h**

7.42 mysqlpp::SQLQuery Class Reference

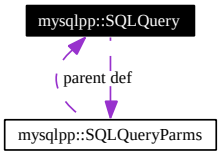
The base class for **mysqlpp::Query** (p. 101).

`#include <sql_query.h>`

Inheritance diagram for mysqlpp::SQLQuery:



Collaboration diagram for mysqlpp::SQLQuery:



Public Methods

- **SQLQuery** ()
  Default constructor.
- **SQLQuery** (const SQLQuery &q)
  Create a query as a copy of another.
- void **parse** ()
  Treat the contents of the query string as a template query.
- std::string **error** () const
  Return the last error message that was set.
- bool **success** () const
  Return true if the last query was successful.
- **operator bool** ()
  Return true if the last query was successful.
- bool **operator!** ()
  Return true if the last query failed.
- void **reset** ()
  Reset the query object so that it can be reused.
- template<class T> SQLQuery & **update** (const T &o, const T &n)
  Build an UPDATE SQL query.

- template<class T> SQLQuery & **insert** (const T &v)

  Build an INSERT SQL query for one record.

- template<class Iter> SQLQuery & **insert** (Iter first, Iter last)

  Build an INSERT SQL query for multiple records.

- template<class T> SQLQuery & **replace** (const T &v)

  Build a REPLACE SQL query.

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

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

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

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

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

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

- std::string **str** (**SQLQueryParms** &p, **query_reset** r)

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

Public Attributes

- **SQLQueryParms def**

  The default template parameters.

Protected Types

- typedef const **SQLString** & **ss**

  to keep parameters lists short

- typedef **SQLQueryParms parms**

  abstraction; remove when **Query** (p. 101) *and* **SQLQuery** (p. 132) *merge*

Protected Methods

- void **proc** (**SQLQueryParms** &p)

  Process a parameterized query list.

Protected Attributes

- bool **Success**

  if true, last query succeeded

- char * **errmsg**

  string explaining last query error

- std::vector< **SQLParseElement** > **parsed**

  List of template query parameters.

- std::vector< std::string > **parsed_names**

  Maps template parameter position values to the corresponding parameter name.

- std::map< std::string, int > **parsed_nums**

  Maps template parameter names to their position value.

7.42.1 Detailed Description

The base class for **mysqlpp::Query** (p. 101).

One uses an object of this class to form queries that can be sent to the database server via the **mysqlpp::Connection** (p. 42) object.

This class is subclassed from `std::stringstream`. This means that you can form a SQL query using C++ stream idioms without having to create your own `stringstream` object and then dump that into the query object. And of course, it gets you all the benefits of C++ streams, such as type safety, which `sprintf()` and such do not offer. Although you can read from this object as you would any other stream, this is *not* recommended. It may fail in strange ways, and there is no support offered if you break it by doing so.

If you seek within the stream in any way, be sure to reset the stream pointer to the end before calling any of the **SQLQuery** (p. 132)-specific methods except for **error()** (p. 134) and **success()** (p. 132).

7.42.2 Member Function Documentation

7.42.2.1 std::string mysqlpp::SQLQuery::error () const [inline]

Return the last error message that was set.

This function has no real meaning at this level. It's overridden with useful behavior in the **Query** (p. 101) subclass.

7.42.2.2 template<class Iter> SQLQuery& mysqlpp::SQLQuery::insert (Iter *first*, Iter *last*) [inline]

Build an INSERT SQL query for multiple records.

Parameters:

first iterator pointing to first SSQLS to insert

last iterator pointing to record past last SSQLS to insert

Reimplemented in **mysqlpp::Query** (p. 103).

7.42.2.3 template<class T> SQLQuery& mysqlpp::SQLQuery::insert (const T & *v*) [inline]

Build an INSERT SQL query for one record.

Parameters:

v SSQLS which will be inserted into `v.table()`

Reimplemented in **mysqlpp::Query** (p. 104).

7.42.2.4 void mysqlpp::SQLQuery::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.

7.42.2.5 template<class T> SQLQuery& mysqlpp::SQLQuery::replace (const T & *v*) [inline]

Build a REPLACE SQL query.

Parameters:

v SSQLS containing data to insert into the table, or which will replace an existing record if this record matches another on a unique index

Reimplemented in **mysqlpp::Query** (p. 104).

7.42.2.6 void mysqlpp::SQLQuery::reset ()

Reset the query object so that it can be reused.

This erases the query string and the contents of the parameterized query element list.

7.42.2.7 std::string mysqlpp::SQLQuery::str (SQLQueryParms & *p*, query_reset *r*)

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

Parameters:

p template query parameters to use, overriding the ones this object holds, if any

r if equal to `RESET_QUERY`, query object is cleared after this call

7.42.2.8 std::string mysqlpp::SQLQuery::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

7.42.2.9 std::string mysqlpp::SQLQuery::str (query_reset *r*) [inline]

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

Parameters:

r if equal to `RESET_QUERY`, query object is cleared after this call

7.42.2.10 template<class T> SQLQuery& mysqlpp::SQLQuery::update (const T & *o*, const T & *n*) [inline]

Build an UPDATE SQL query.

Parameters:

o SSQLS containing data to match in table

n SSQLS that will replace any records in `o.table()` that match `o`

Reimplemented in **mysqlpp::Query** (p. 105).

7.42.3 Member Data Documentation

7.42.3.1 SQLQueryParms mysqlpp::SQLQuery::def

The default template parameters.

Used for filling in parameterized queries.

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

- **sql_query.h**
- sql_query.cpp

7.43 mysqlpp::SQLQueryNEParms Class Reference

Exception thrown when not enough parameters are provided.

`#include <exceptions.h>`

Public Methods

- **SQLQueryNEParms** (const char *c)
  Create exception object.
- **~SQLQueryNEParms** () throw ()
  Destroy exception object.
- virtual const char * **what** () const throw ()
  Returns the error message.

Public Attributes

- const char * **error**
  MySQL error string.

7.43.1 Detailed Description

Exception thrown when not enough parameters are provided.

Thrown when not enough parameters are provided for a template query.

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

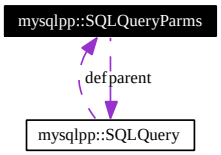
- **exceptions.h**

7.44 mysqlpp::SQLQueryParms Class Reference

This class holds the parameter values for filling template queries.

`#include <sql_query.h>`

Collaboration diagram for mysqlpp::SQLQueryParms:



Public Methods

- **SQLQueryParms** ()
  Default constructor.
- **SQLQueryParms** (**SQLQuery** *p)
  Create object.
- bool **bound** ()
  Returns true if we are bound to a query object.
- void **clear** ()
  Clears the list.
- **SQLString** & **operator**[] (size_type n)
  Access element number n.
- const **SQLString** & **operator**[] (size_type n) const
  Access element number n.
- **SQLString** & **operator**[] (const char *str)
  Access the value of the element with a key of str.
- const **SQLString** & **operator**[] (const char *str) const
  Access the value of the element with a key of str.
- SQLQueryParms & **operator**<< (const **SQLString** &str)
  Adds an element to the list.
- SQLQueryParms & **operator**+= (const **SQLString** &str)
  Adds an element to the list.
- SQLQueryParms **operator**+ (const SQLQueryParms &other) const
  Build a composite of two parameter lists.
- void **set** (ss a, ss b, ss c, ss d, ss e, ss f, ss g, ss h, ss i, ss j, ss k, ss l)
  Set (p. 129) *the template query parameters.*

7.44.1 Detailed Description

This class holds the parameter values for filling template queries.

7.44.2 Constructor & Destructor Documentation

7.44.2.1 mysqlpp::SQLQueryParms::SQLQueryParms (SQLQuery * *p*) [inline]

Create object.

Parameters:
p pointer to the query object these parameters are tied to

7.44.3 Member Function Documentation

7.44.3.1 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.

7.44.3.2 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.

7.44.3.3 void mysqlpp::SQLQueryParms::set (ss *a*, ss *b*, ss *c*, ss *d*, ss *e*, ss *f*, ss *g*, ss *h*, ss *i*, ss *j*, ss *k*, ss *l*) [inline]

Set (p. 129) 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:

- **sql_query.h**
- sql_query.cpp

7.45 mysqlpp::SQLString Class Reference

A specialized `std::string` that will convert from any valid MySQL type.

`#include <sql_string.h>`

Public Methods

- **SQLString** ()
  Default constructor; empty string.
- **SQLString** (const std::string &str)
  Create object as a copy of a C++ string.
- **SQLString** (const char *str)
  Create object as a copy of a C string.
- **SQLString** (char i)
  Create object as the string form of a `char` *value.*
- **SQLString** (unsigned char i)
  Create object as the string form of an `unsigned char` *value.*
- **SQLString** (short int i)
  Create object as the string form of a `short int` *value.*
- **SQLString** (unsigned short int i)
  Create object as the string form of an `unsigned short int` *value.*
- **SQLString** (int i)
  Create object as the string form of an `int` *value.*
- **SQLString** (unsigned int i)
  Create object as the string form of an `unsigned int` *value.*
- **SQLString** (**longlong** i)
  Create object as the string form of a **longlong** *value.*
- **SQLString** (**ulonglong** i)
  Create object as the string form of an `unsigned` **longlong** *value.*
- **SQLString** (float i)
  Create object as the string form of a `float` *value.*
- **SQLString** (double i)
  Create object as the string form of a `double` *value.*
- SQLString & **operator=** (const char *str)
  Copy a C string into this object.

- SQLString & **operator=** (const std::string &str)

  Copy a C++ `string` *into this object.*

Public Attributes

- bool **is_string**

  If true, the object's string data is a copy of another string. Otherwise, it's the string form of an integral type.

- bool **dont_escape**

  If true, the string data doesn't need to be SQL-escaped when building a query.

- bool **processed**

  If true, one of the MySQL++ manipulators has processed the string data.

7.45.1 Detailed Description

A specialized `std::string` that will convert from any valid MySQL type.

7.45.2 Member Data Documentation

7.45.2.1 bool mysqlpp::SQLString::processed

If true, one of the MySQL++ manipulators has processed the string data.

"Processing" is escaping special SQL characters, and/or adding quotes. See the documentation for **manip.h** for details.

This flag is used by the template query mechanism, to prevent a string from being re-escaped or re-quoted each time that query is reused.

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

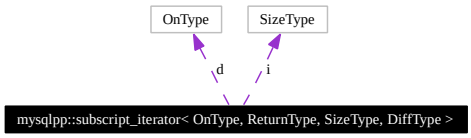
- **sql_string.h**
- sql_string.cpp

7.46 mysqlpp::subscript_iterator< OnType, ReturnType, SizeType, DiffType > Class Template Reference

Iterator that can be subscripted.

`#include <resiter.h>`

Collaboration diagram for mysqlpp::subscript_iterator< OnType, ReturnType, SizeType, DiffType >:



Public Methods

- **subscript_iterator** ()

  Default constructor.

- **subscript_iterator** (OnType *what, SizeType pos)

  Create iterator given the container and a position within it.

- bool **operator==** (const subscript_iterator &j) const

  Return true if given iterator points to the same container and the same position within the container.

- bool **operator!=** (const subscript_iterator &j) const

  Return true if given iterator is different from this one, but points to the same container.

- bool **operator<** (const subscript_iterator &j) const

  Return true if the given iterator points to the same container as this one, and that this iterator's position is less than the given iterator's.

- bool **operator>** (const subscript_iterator &j) const

  Return true if the given iterator points to the same container as this one, and that this iterator's position is greater than the given iterator's.

- bool **operator<=** (const subscript_iterator &j) const

  Return true if the given iterator points to the same container as this one, and that this iterator's position is less than or equal to the given iterator's.

- bool **operator>=** (const subscript_iterator &j) const

  Return true if the given iterator points to the same container as this one, and that this iterator's position is greater than or equal to the given iterator's.

- ReturnType * **operator** -> () const

  Access the current pointed-to element within the container.

- ReturnType **operator** * () const

Dereference the iterator, returning the pointed-to element within the container.

- ReturnType **operator**[] (SizeType n) const

  Return the an element in the container given its index.

- subscript_iterator & **operator++** ()

  Move the iterator to the next element, returning an iterator to that element.

- subscript_iterator **operator++** (int)

  Move the iterator to the next element, returning an iterator to the element we were pointing at before the change.

- subscript_iterator & **operator–** ()

  Move the iterator to the previous element, returning an iterator to that element.

- subscript_iterator **operator–** (int)

  Move the iterator to the previous element, returning an iterator to the element we were pointing at before the change.

- subscript_iterator & **operator+=** (SizeType n)

  Advance iterator position by `n`.

- subscript_iterator **operator+** (SizeType n) const

  Return an iterator `n` *positions beyond this one.*

- subscript_iterator & **operator-=** (SizeType n)

  Move iterator position back by `n`.

- subscript_iterator **operator-** (SizeType n) const

  Return an iterator `n` *positions before this one.*

- DiffType **operator-** (const subscript_iterator &j) const

  Return an iterator `n` *positions before this one.*

7.46.1 Detailed Description

template<class OnType, class ReturnType, class SizeType, class DiffType> class mysqlpp::subscript_iterator< OnType, ReturnType, SizeType, DiffType >

Iterator that can be subscripted.

This is the type of iterator used by the **const_subscript_container** (p. 57) template.

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

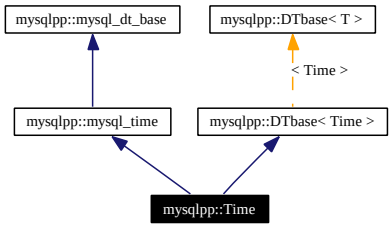
- **resiter.h**

7.47 mysqlpp::Time Struct Reference

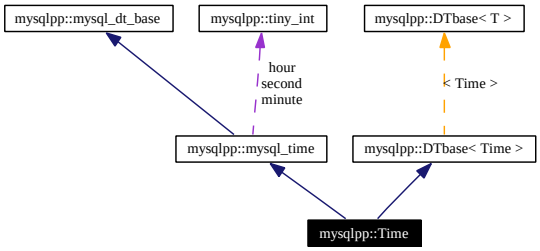
Holds MySQL times.

`#include <datetime.h>`

Inheritance diagram for mysqlpp::Time:



Collaboration diagram for mysqlpp::Time:



Public Methods

- **Time** ()

  Default constructor.

- **Time** (**cchar** *str)

  Initialize object from a MySQL time string.

- **Time** (const **ColData** &str)

  Initialize object from a MySQL time string.

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

  Initialize object from a MySQL time string.

- short int **compare** (const Time &other) const

  Compare this time to another.

7.47.1 Detailed Description

Holds MySQL times.

Objects of this class can be inserted into streams, and initialized from MySQL TIME strings.

7.47.2 Constructor & Destructor Documentation

7.47.2.1 mysqlpp::Time::Time (cchar * *str*) [inline]

Initialize object from a MySQL time string.

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

7.47.2.2 mysqlpp::Time::Time (const ColData & *str*) [inline]

Initialize object from a MySQL time string.

See also:
Time(cchar*) (p. 145)

7.47.2.3 mysqlpp::Time::Time (const std::string & *str*) [inline]

Initialize object from a MySQL time string.

See also:
Time(cchar*) (p. 145)

7.47.3 Member Function Documentation

7.47.3.1 short int mysqlpp::Time::compare (const Time & *other*) const [inline, virtual]

Compare this time to another.

See also:
mysql_time::compare() (p. 84) for implementation

Implements **mysqlpp::DTbase< Time >** (p. 71).

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

- **datetime.h**

7.48 mysqlpp::tiny_int Class Reference

Class for holding an SQL `tiny_int` object.

`#include <tiny_int.h>`

Public Methods

- **tiny_int** ()
  Default constructor.
- **tiny_int** (short int v)
  Create object from any integral type that can be converted to a `short int`.
- **operator short int** () const
  Return value as a `short int`.
- tiny_int & **operator=** (short int v)
  Assign a `short int` *to the object.*
- tiny_int & **operator+=** (short int v)
  Add another value to this object.
- tiny_int & **operator-=** (short int v)
  Subtract another value to this object.
- tiny_int & **operator** *= (short int v)
  Multiply this value by another object.
- tiny_int & **operator/=** (short int v)
  Divide this value by another object.
- tiny_int & **operator%=** (short int v)
  Divide this value by another object and store the remainder.
- tiny_int & **operator &=** (short int v)
  Bitwise AND this value by another value.
- tiny_int & **operator|=** (short int v)
  Bitwise OR this value by another value.
- tiny_int & **operator^=** (short int v)
  Bitwise XOR this value by another value.
- tiny_int & **operator<<=** (short int v)
  Shift this value left by `v` *positions.*
- tiny_int & **operator>>=** (short int v)
  Shift this value right by `v` *positions.*

- tiny_int & **operator++** ()
  Add one to this value and return that value.
- tiny_int & **operator–** ()
  Subtract one from this value and return that value.
- tiny_int **operator++** (int)
  Add one to this value and return the previous value.
- tiny_int **operator–** (int)
  Subtract one from this value and return the previous value.
- tiny_int **operator-** (const tiny_int &i) const
  Return this value minus `i`.
- tiny_int **operator+** (const tiny_int &i) const
  Return this value plus `i`.
- tiny_int **operator** * (const tiny_int &i) const
  Return this value multiplied by `i`.
- tiny_int **operator/** (const tiny_int &i) const
  Return this value divided by `i`.
- tiny_int **operator%** (const tiny_int &i) const
  Return the modulus of this value divided by `i`.
- tiny_int **operator|** (const tiny_int &i) const
  Return this value bitwise OR'd by `i`.
- tiny_int **operator &** (const tiny_int &i) const
  Return this value bitwise AND'd by `i`.
- tiny_int **operator^** (const tiny_int &i) const
  Return this value bitwise XOR'd by `i`.
- tiny_int **operator<<** (const tiny_int &i) const
  Return this value bitwise shifted left by `i`.
- tiny_int **operator>>** (const tiny_int &i) const
  Return this value bitwise shifted right by `i`.

7.48.1 Detailed Description

Class for holding an SQL `tiny_int` object.

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.

Several of the functions below accept a `short int` argument, but internally we store the data as a `char`. Beware of integer overflows!

7.48.2 Constructor & Destructor Documentation

7.48.2.1 mysqlpp::tiny_int::tiny_int () [inline]

Default constructor.

Value is uninitialized

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

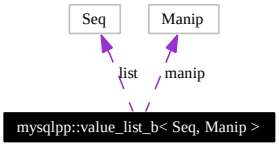
- **tiny_int.h**

7.49 mysqlpp::value_list_b< Seq, Manip > Struct Template Reference

Same as **value_list_ba** (p. 151), plus the option to have some elements of the list suppressed.

`#include <vallist.h>`

Collaboration diagram for mysqlpp::value_list_b< Seq, Manip >:



Public Methods

- **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 * **delem**

  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

7.49.1 Detailed Description

template<class Seq, class Manip> struct mysqlpp::value_list_b< Seq, Manip >

Same as **value_list_ba** (p. 151), 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** (p. 151)'s documentation for more details.

7.49.2 Constructor & Destructor Documentation

7.49.2.1 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:

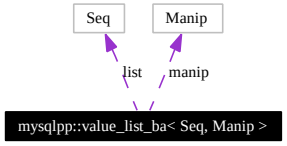
- **vallist.h**

7.50 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 Methods

- **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 * **delem**
  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

7.50.1 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()** (p. 30) functions build instances of this structure template to do their work. MySQL++'s SSQLS mechanism calls those functions when building SQL queries; you can call them yourself to do similar work. The "Harnessing SSQLS Internals" section of the user manual has some examples of this.

See also:
value_list_b (p. 149)

7.50.2 Constructor & Destructor Documentation

7.50.2.1 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**

Chapter 8

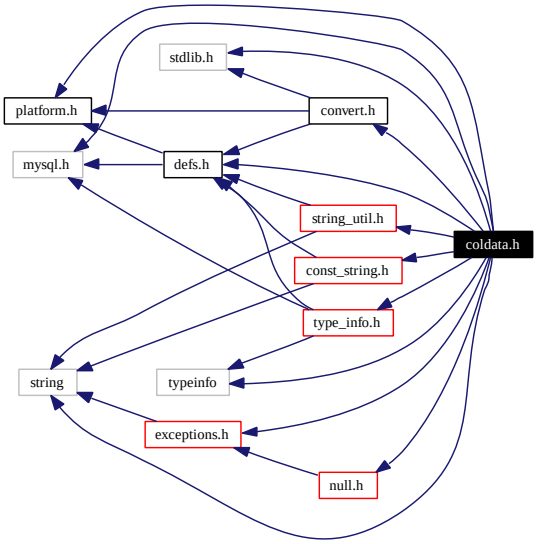
MySQL++ File Documentation

8.1 coldata.h File Reference

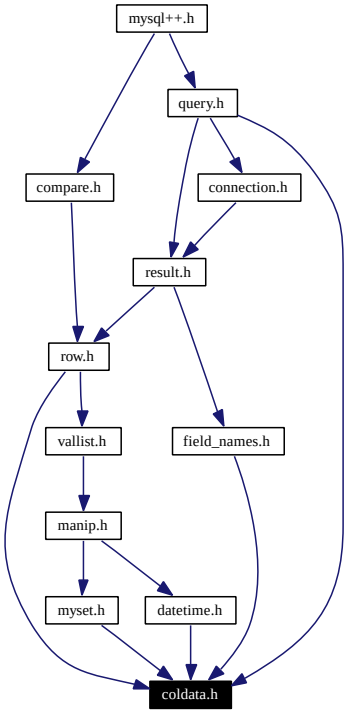
Declares classes for converting string data to any of the basic C types.

```
#include "platform.h"
#include "const_string.h"
#include "convert.h"
#include "defs.h"
#include "exceptions.h"
#include "null.h"
#include "string_util.h"
#include "type_info.h"
#include <mysql.h>
#include <typeinfo>
#include <string>
#include <stdlib.h>
```

Include dependency graph for coldata.h:



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



Namespaces

- namespace **mysqlpp**

8.1.1 Detailed Description

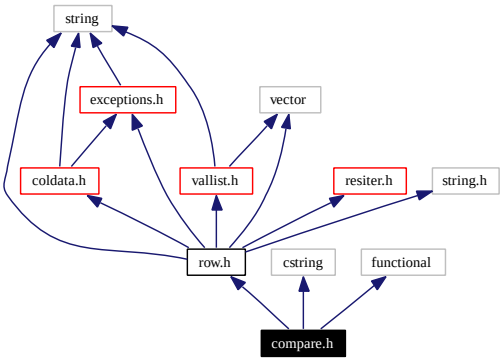
Declares classes for converting string data to any of the basic C types.

Roughly speaking, this defines classes that are the inverse of **mysqlpp::SQLString** (p. 140).

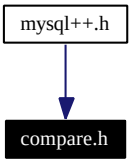
8.2 compare.h File Reference

```
#include "row.h"
#include <cstring>
#include <functional>
```

Include dependency graph for compare.h:



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



Namespaces

- namespace **mysqlpp**

8.2.1 Detailed Description

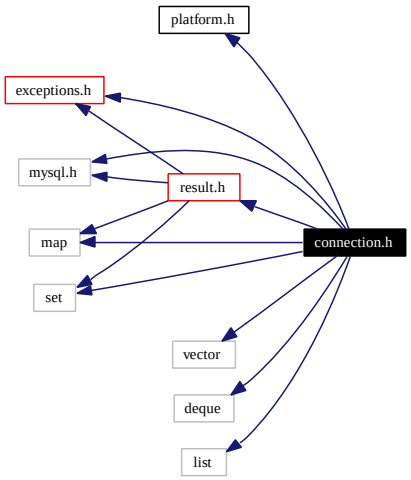
Declares several function objects and templates for creating function objects for comparing various things. These are useful when using STL algorithms like std::find_if() on containers of data retreived from a database with MySQL++.

8.3 connection.h File Reference

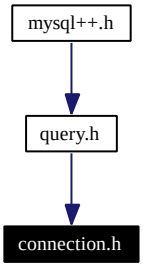
Declares the Connection class.

```
#include "platform.h"
#include "exceptions.h"
#include "result.h"
#include <mysql.h>
#include <vector>
#include <deque>
#include <list>
#include <set>
#include <map>
```

Include dependency graph for connection.h:



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



Namespaces

- namespace **mysqlpp**

8.3.1 Detailed Description

Declares the Connection class.

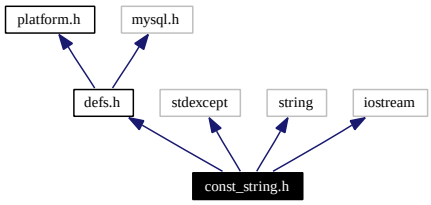
Every program using MySQL++ must create a Connection object, which manages information about the connection to the MySQL database. In addition, this class controls things like whether exceptions are thrown when errors are encountered.

8.4 const_string.h File Reference

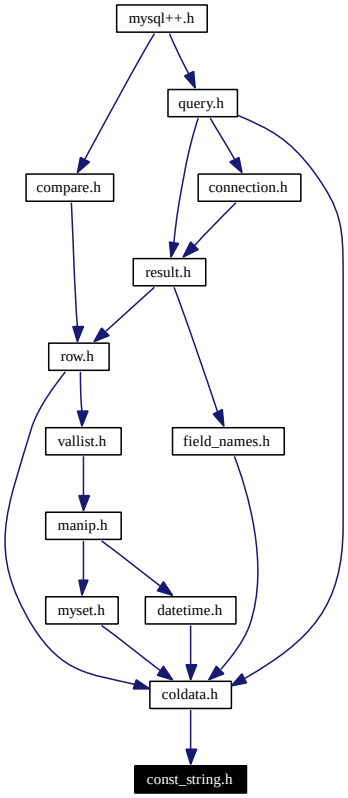
Declares a wrapper for `const char*` which behaves in a way more useful to MySQL++.

```
#include "defs.h"
#include <stdexcept>
#include <string>
#include <iostream>
```

Include dependency graph for const_string.h:



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



Namespaces

- namespace **mysqlpp**

8.4.1 Detailed Description

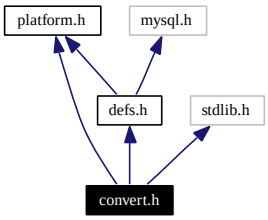
Declares a wrapper for `const char*` which behaves in a way more useful to MySQL++.

8.5 convert.h File Reference

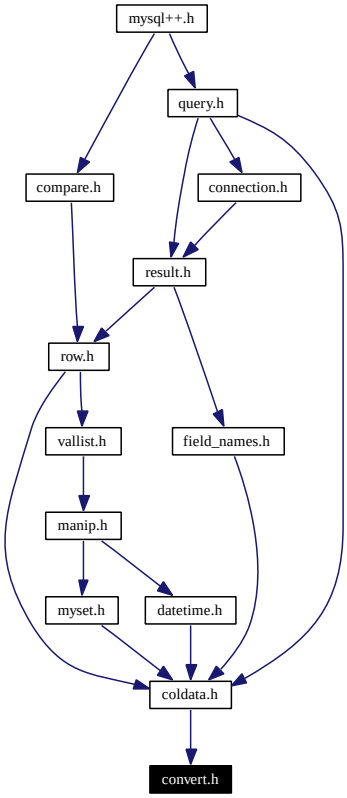
Declares various string-to-integer type conversion templates.

```
#include "platform.h"
#include "defs.h"
#include <stdlib.h>
```

Include dependency graph for convert.h:



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



Namespaces

- namespace **mysqlpp**

8.5.1 Detailed Description

Declares various string-to-integer type conversion templates.

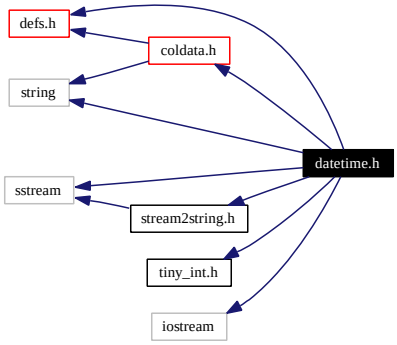
These templates are the mechanism used within **mysqlpp::ColData_Tmpl** (p. 38) for its string-to-*something* conversions.

8.6 datetime.h File Reference

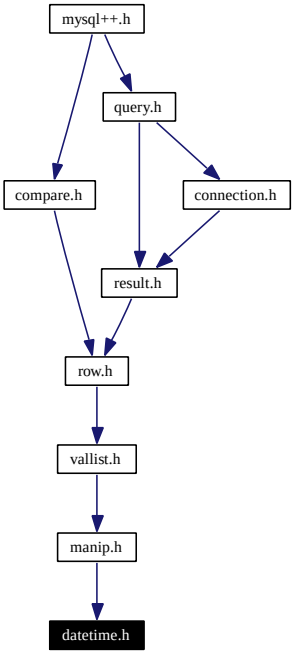
Declares classes to add MySQL-compatible date and time types to C++'s type system.

```
#include "defs.h"
#include "coldata.h"
#include "stream2string.h"
#include "tiny_int.h"
#include <string>
#include <sstream>
#include <iostream>
```

Include dependency graph for datetime.h:



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



Namespaces

- namespace **mysqlpp**

8.6.1 Detailed Description

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

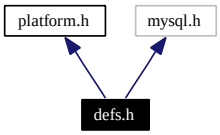
8.7 defs.h File Reference

Standard definitions used all across the library, particularly things that don't fit well anywhere else.

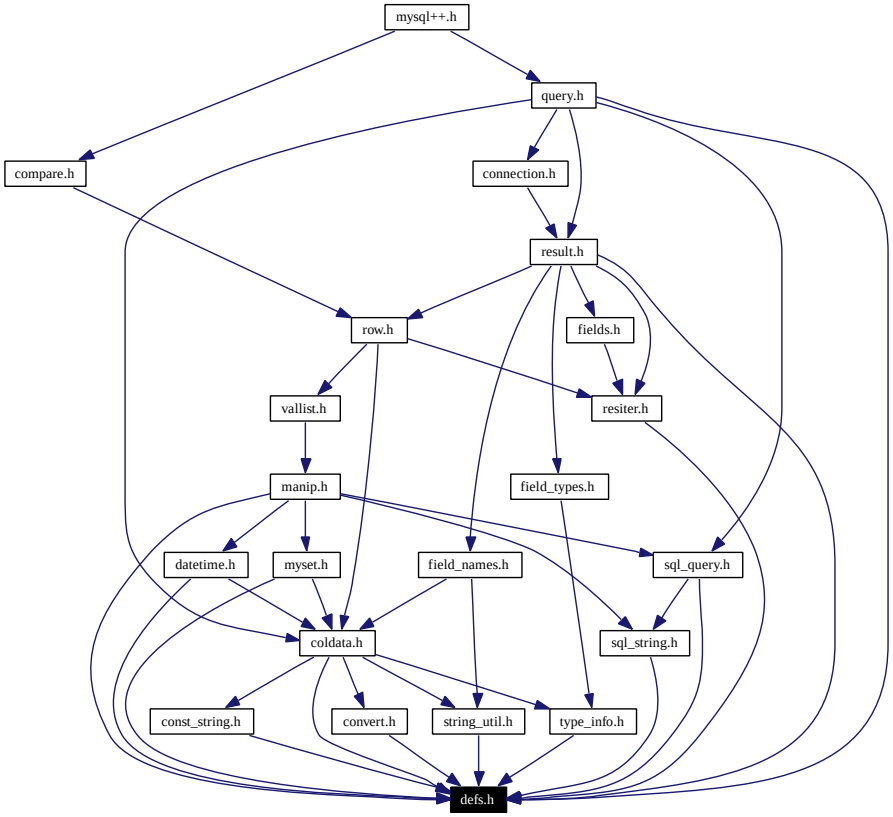
```
#include "platform.h"
```

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

Include dependency graph for defs.h:



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



Namespaces

- namespace **mysqlpp**

8.7.1 Detailed Description

Standard definitions used all across the library, particularly things that don't fit well anywhere else.

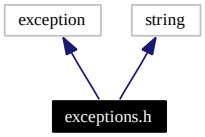
8.8 exceptions.h File Reference

Declares the MySQL++-specific exception classes.

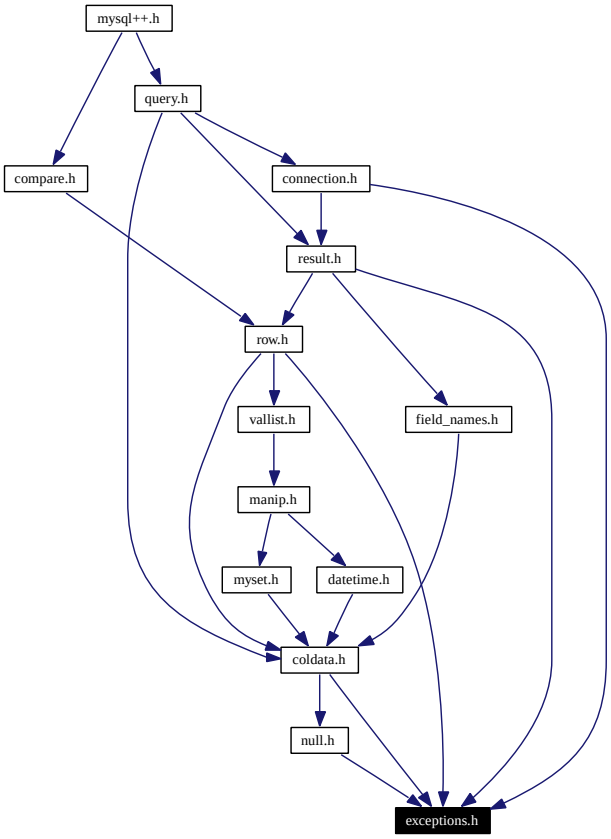
`#include <exception>`

`#include <string>`

Include dependency graph for exceptions.h:



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



Namespaces

- namespace **mysqlpp**

8.8.1 Detailed Description

Declares the MySQL++-specific exception classes.

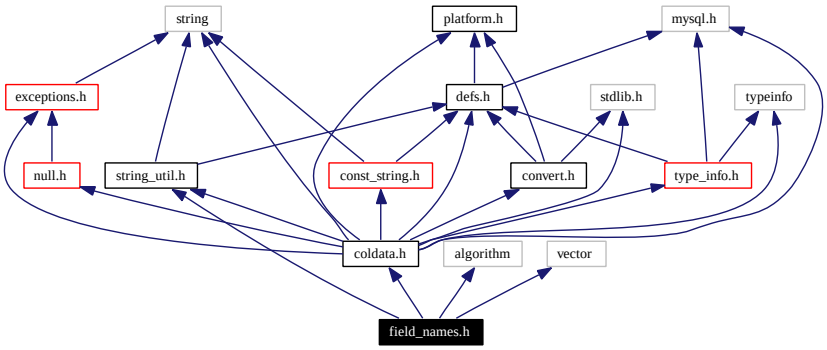
When exceptions are enabled for a given **Connection** (p. 42) object, any of these exceptions can be thrown as a result of operations done through that connection.

8.9 field_names.h File Reference

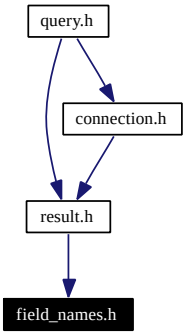
Declares a class to hold a list of field names.

```
#include "coldata.h"
#include "string_util.h"
#include <algorithm>
#include <vector>
```

Include dependency graph for field_names.h:



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



Namespaces

- namespace **mysqlpp**

8.9.1 Detailed Description

Declares a class to hold a list of field names.

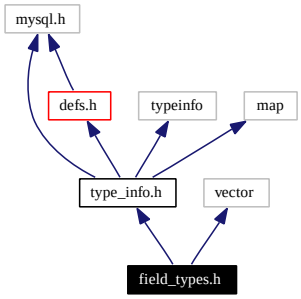
8.10 field_types.h File Reference

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

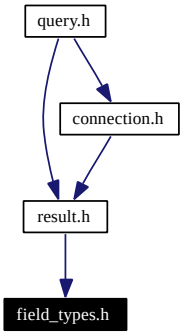
```
#include "type_info.h"
```

```
#include <vector>
```

Include dependency graph for field_types.h:



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



Namespaces

- namespace **mysqlpp**

8.10.1 Detailed Description

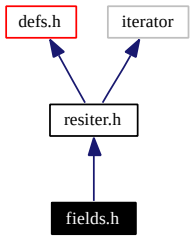
Declares a class to hold a list of SQL field type info.

8.11 fields.h File Reference

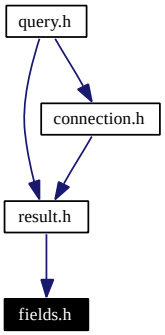
Declares a class for holding information about a set of fields.

`#include "resiter.h"`

Include dependency graph for fields.h:



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



Namespaces

- namespace **mysqlpp**

8.11.1 Detailed Description

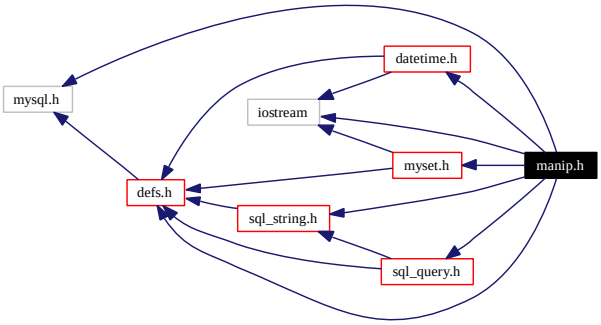
Declares a class for holding information about a set of fields.

8.12 manip.h File Reference

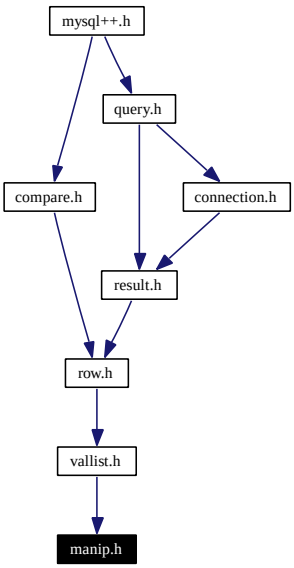
Declares `std::ostream` manipulators useful with SQL syntax.

```
#include "defs.h"
#include "datetime.h"
#include "myset.h"
#include "sql_string.h"
#include "sql_query.h"
#include <mysql.h>
#include <iostream>
```

Include dependency graph for manip.h:



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



Namespaces

- namespace **mysqlpp**

8.12.1 Detailed Description

Declares `std::ostream` manipulators useful with SQL syntax.

These manipulators let you automatically quote elements or escape characters that are special in SQL when inserting them into an `std::ostream`. Since **mysqlpp::SQLQuery** (p. 132) is an ostream, these manipulators make it easier to build syntactically-correct SQL queries.

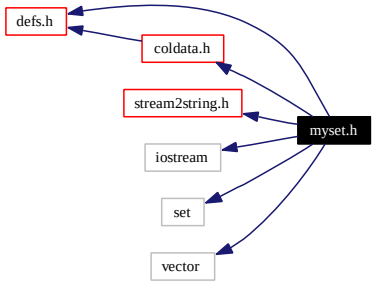
This file also includes `operator<<` definitions for ColData_Tmpl, one of the MySQL++ string-like classes. When inserting such items into a stream, they are automatically quoted and escaped as necessary unless the global variable dont_quote_auto is set to true. These operators are smart enough to turn this behavior off when the stream is `cout` or `cerr`, however, since quoting and escaping are surely not required in that instance.

8.13 myset.h File Reference

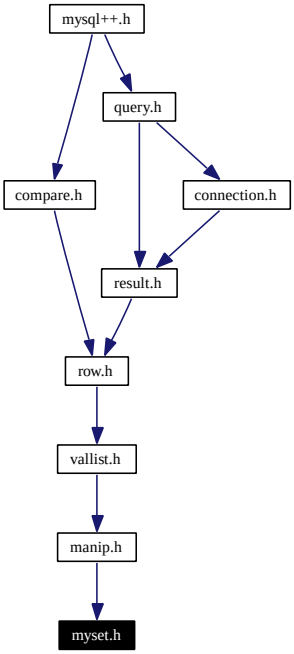
Declares templates for generating custom containers used elsewhere in the library.

```
#include "defs.h"
#include "coldata.h"
#include "stream2string.h"
#include <iostream>
#include <set>
#include <vector>
```

Include dependency graph for myset.h:



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



Namespaces

- namespace **mysqlpp**

8.13.1 Detailed Description

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

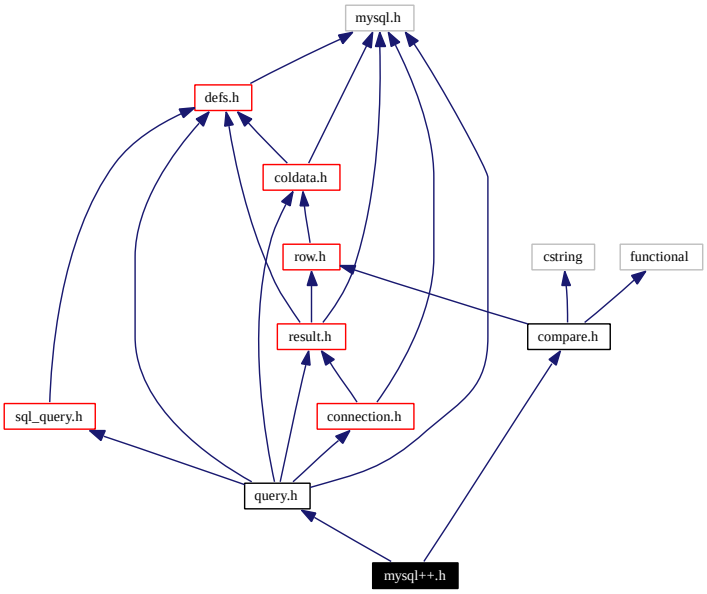
8.14 mysql++.h File Reference

The main MySQL++ header file.

```
#include "query.h"
```

```
#include "compare.h"
```

Include dependency graph for mysql++.h:



8.14.1 Detailed Description

The main MySQL++ header file.

This file brings in all MySQL++ headers except for custom*.h, which is a strictly optional feature of MySQL++.

There is no point in trying to optimize which headers you include, because every MySQL++ program needs **query.h**, and that includes all the other headers indirectly, except for custom*.h and **compare.h**. (**query.h** doesn't bring in **compare.h** because it's not used within the library anywhere; its facilities are only for end-user programs.) The only possible optimization is to include **query.h** instead of **mysql++.h**, and this results only in trivial compile time reductions at the expense of code clarity.

8.15 mysql++.hh File Reference

Deprecated backwards-compatibility header. Use **mysql++.h** in new code instead.

8.15.1 Detailed Description

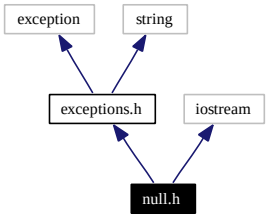
Deprecated backwards-compatibility header. Use **mysql++.h** in new code instead.

8.16 null.h File Reference

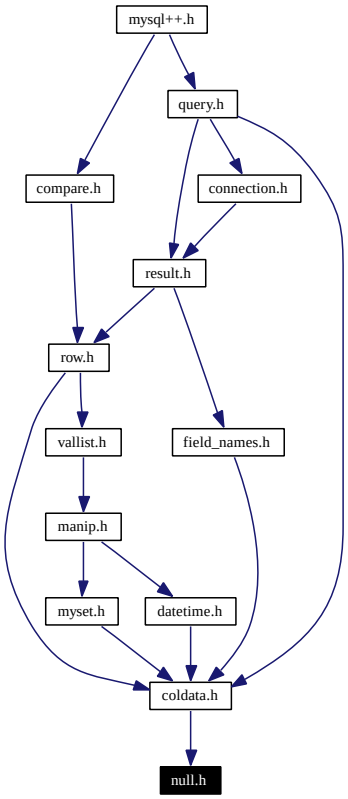
Declares classes that implement SQL "null" semantics within C++'s type system.

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

Include dependency graph for null.h:



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



Namespaces

- namespace **mysqlpp**

8.16.1 Detailed Description

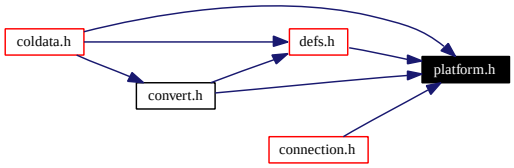
Declares classes that implement SQL "null" semantics within C++'s type system.

This is required because C++'s own NULL type is not semantically the same as SQL nulls.

8.17 platform.h File Reference

This file includes things that help the rest of MySQL++.

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



8.17.1 Detailed Description

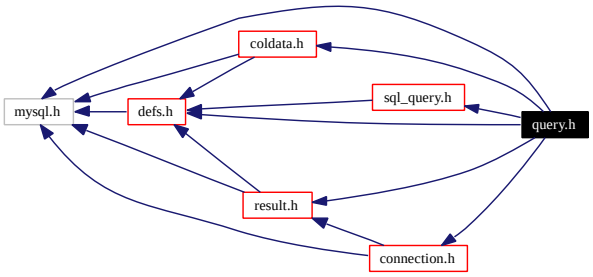
This file includes things that help the rest of MySQL++.

8.18 query.h File Reference

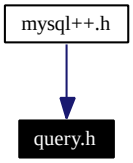
Defines the user-facing Query class, which is used to build up SQL queries, and execute them.

```
#include "defs.h"
#include "coldata.h"
#include "connection.h"
#include "result.h"
#include "sql_query.h"
#include <mysql.h>
```

Include dependency graph for query.h:



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



Namespaces

- namespace **mysqlpp**

Defines

- #define **mysql_query_define1**(RETURN, FUNC)

  Used to define many similar member functions in class Query.

- #define **mysql_query_define2**(FUNC)

  Used to define many similar member functions in class Query.

8.18.1 Detailed Description

Defines the user-facing Query class, which is used to build up SQL queries, and execute them.

This class is used in one of two main fashions.

First, it has several member functions that can build specific query types. For instance, **insert()** (p. 104) allows you to build an INSERT statement for a Specialized SQL Structure.

Second, one of its base classes is std::stringstream, so you may build an SQL query in the same way as you'd build up any other string with C++ streams.

One does not generally create Query objects directly. Instead, call **mysqlpp::Connection::query()** (p. 50) to get one tied to that connection.

8.18.2 Define Documentation

8.18.2.1 #define mysql_query_define1(RETURN, FUNC)

Value:

```
RETURN FUNC (const char* str); \
  RETURN FUNC (parms &p);\
  mysql_query_define0(RETURN,FUNC) \
```

Used to define many similar member functions in class Query.

8.18.2.2 #define mysql_query_define2(FUNC)

Value:

```
template <class T1> void FUNC (T1 &con, const char* str); \
  template <class T1> void FUNC (T1 &con, parms &p, query_reset r = RESET_QUERY);\
  template <class T1> void FUNC (T1 &con, ss a)\
        {FUNC (con, parms() << a);}\
  template <class T1> void FUNC (T1 &con, ss a, ss b)\
        {FUNC (con, parms() << a << b);}\
  template <class T1> void FUNC (T1 &con, ss a, ss b, ss c)\
        {FUNC (con, parms() << a << b << c);}\
  template <class T1> void FUNC (T1 &con, ss a, ss b, ss c, ss d)\
        {FUNC (con, parms() << a << b << c << d);}\
  template <class T1> void FUNC (T1 &con, ss a, ss b, ss c, ss d, ss e)\
        {FUNC (con, parms() << a << b << c << d << e);} \
  template <class T1> void FUNC (T1 &con, ss a, ss b, ss c, ss d, ss e, ss f)\
        {FUNC (con, parms() << a << b << c << d << e << f);}\
  template <class T1> void FUNC (T1 &con,ss a,ss b,ss c,ss d,ss e,ss f,ss g)\
        {FUNC (con, parms() << a << b << c << d << e << f << g);}\
  template <class T1> void FUNC (T1 &con,ss a,ss b,ss c,ss d,ss e,ss f,ss g,ss h)\
        {FUNC (con, parms() << a << b << c << d << e << f << g << h);}\
  template <class T1> void FUNC (T1 &con, ss a, ss b, ss c, ss d, ss e, ss f, ss g, ss h, ss i)\
        {FUNC (con, parms() << a << b << c << d << e << f << g << h << i);}\
  template <class T1> void FUNC (T1 &con, ss a,ss b,ss c,ss d,ss e,ss f,ss g,ss h,ss i,ss j)\
        {FUNC (con, parms() <<a <<b <<c <<d <<e <<f <<g <<h <<i <<j);}\
  template <class T1> void FUNC (T1 &con, ss a,ss b,ss c,ss d,ss e,ss f,ss g,ss h,ss i,ss j,ss k)\
        {FUNC (con, parms() <<a <<b <<c <<d <<e <<f <<g <<h <<i <<j <<k);}\
  template <class T1> void FUNC (T1 &con, ss a,ss b,ss c,ss d,ss e,ss f,ss g,ss h,ss i,ss j,ss k,\
                          ss l)\
        {FUNC (con, parms() <<a <<b <<c <<d <<e <<f <<g <<h <<i <<j <<k <<l);}\
```

Used to define many similar member functions in class Query.

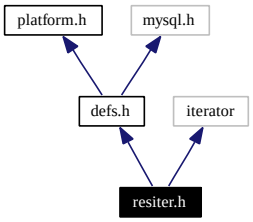
8.19 resiter.h File Reference

Declares templates for adapting existing classes to be iteratable random-access containers.

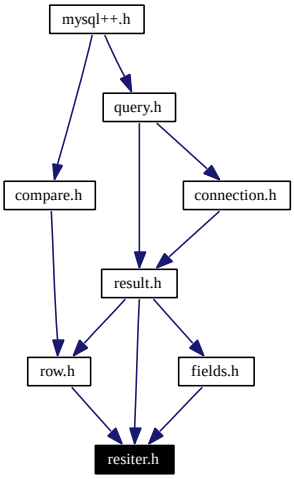
```
#include "defs.h"
```

```
#include <iterator>
```

Include dependency graph for resiter.h:



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



Namespaces

- namespace **mysqlpp**

8.19.1 Detailed Description

Declares templates for adapting existing classes to be iteratable random-access containers.

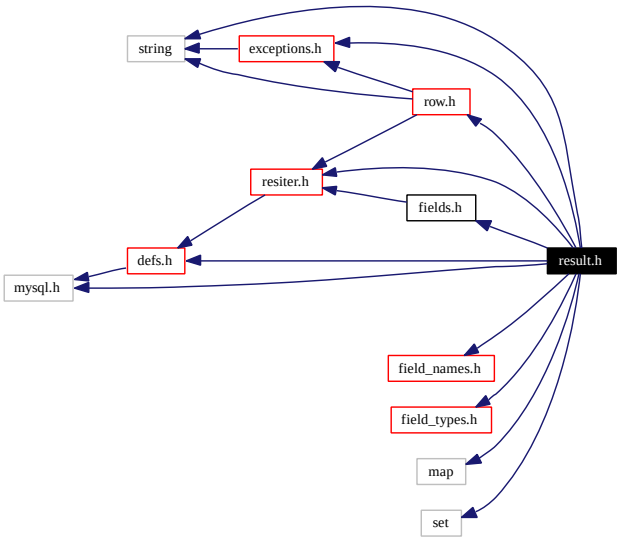
The file name seems to tie it to the **mysqlpp::Result** (p. 108) class, which is so adapted, but these templates are also used to adapt the **mysqlpp::Fields** (p. 77) and **mysqlpp::Row** (p. 116) classes.

8.20 result.h File Reference

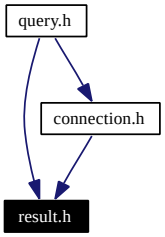
Declares classes for holding SQL query result sets.

```
#include "defs.h"
#include "exceptions.h"
#include "fields.h"
#include "field_names.h"
#include "field_types.h"
#include "resiter.h"
#include "row.h"
#include <mysql.h>
#include <map>
#include <set>
#include <string>
```

Include dependency graph for result.h:



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



Namespaces

- namespace **mysqlpp**

8.20.1 Detailed Description

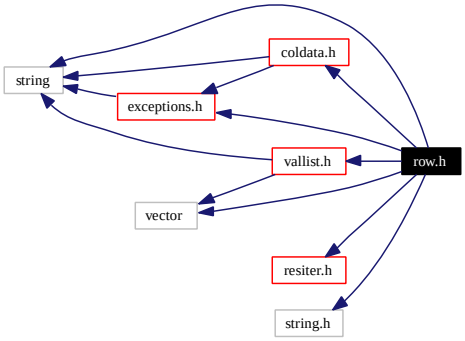
Declares classes for holding SQL query result sets.

8.21 row.h File Reference

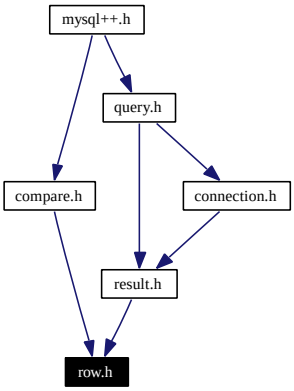
Declares the classes for holding row data from a result set.

```
#include "coldata.h"
#include "exceptions.h"
#include "resiter.h"
#include "vallist.h"
#include <vector>
#include <string>
#include <string.h>
```

Include dependency graph for row.h:



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



Namespaces

- namespace **mysqlpp**

8.21.1 Detailed Description

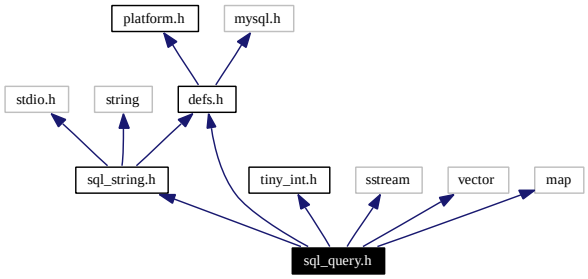
Declares the classes for holding row data from a result set.

8.22 sql_query.h File Reference

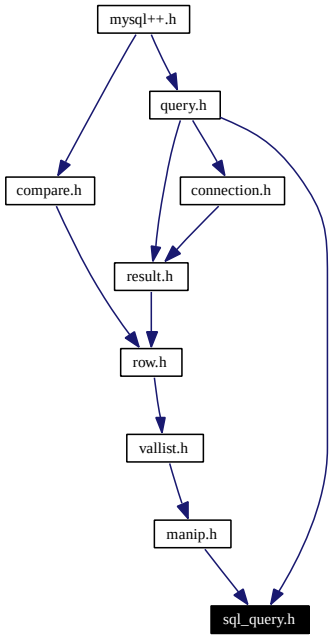
Declares the base class for **mysqlpp::Query** (p. 101), plus some utility classes to be used with it.

```
#include "defs.h"
#include "sql_string.h"
#include "tiny_int.h"
#include <sstream>
#include <vector>
#include <map>
```

Include dependency graph for sql_query.h:



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



Namespaces

- namespace **mysqlpp**

Defines

- #define **mysql_query_define0**(RETURN, FUNC)

  Used to define many similar functions in class SQLQuery.

8.22.1 Detailed Description

Declares the base class for **mysqlpp::Query** (p. 101), plus some utility classes to be used with it.

Class SQLQuery contains a large part of the functionality of class Query, which is the only thing that derives from this class. It is separate for historical reasons only: early on, there was a dream (and some effort) to make MySQL++ database-independent. Once maintainership shifted to MySQL AB employees in 1999, though, that dream died.

The current maintainers have no wish to try and revive that dream, so at some point this class's contents will be folded into the Query class. This will probably happen in the next major release, when major ABI breakage is acceptable.

8.22.2 Define Documentation

8.22.2.1 #define mysql_query_define0(RETURN, FUNC)

Value:

```
RETURN FUNC (ss a)\
    {return FUNC (parms() << a);}\
  RETURN FUNC (ss a, ss b)\
    {return FUNC (parms() << a << b);}\
  RETURN FUNC (ss a, ss b, ss c)\
    {return FUNC (parms() << a << b << c);}\
  RETURN FUNC (ss a, ss b, ss c, ss d)\
    {return FUNC (parms() << a << b << c << d);}\
  RETURN FUNC (ss a, ss b, ss c, ss d, ss e)\
    {return FUNC (parms() << a << b << c << d << e);} \
  RETURN FUNC (ss a, ss b, ss c, ss d, ss e, ss f)\
    {return FUNC (parms() << a << b << c << d << e << f);}\
  RETURN FUNC (ss a, ss b, ss c, ss d, ss e, ss f, ss g)\
    {return FUNC (parms() << a << b << c << d << e << f << g);}\
  RETURN FUNC (ss a, ss b, ss c, ss d, ss e, ss f, ss g, ss h)\
    {return FUNC (parms() << a << b << c << d << e << f << g << h);}\
  RETURN FUNC (ss a, ss b, ss c, ss d, ss e, ss f, ss g, ss h, ss i)\
    {return FUNC (parms() << a << b << c << d << e << f << g << h << i);}\
  RETURN FUNC (ss a,ss b,ss c,ss d,ss e,ss f,ss g,ss h,ss i,ss j)\
    {return FUNC (parms() <<a <<b <<c <<d <<e <<f <<g <<h <<i <<j);}\
  RETURN FUNC (ss a,ss b,ss c,ss d,ss e,ss f,ss g,ss h,ss i,ss j,ss k)\
    {return FUNC (parms() <<a <<b <<c <<d <<e <<f <<g <<h <<i <<j <<k);}\
  RETURN FUNC (ss a,ss b,ss c,ss d,ss e,ss f,ss g,ss h,ss i,ss j,ss k,\
                      ss l)\
    {return FUNC (parms() <<a <<b <<c <<d <<e <<f <<g <<h <<i <<j <<k <<l);}\
```

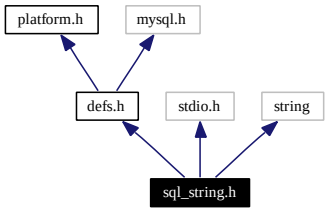
Used to define many similar functions in class SQLQuery.

8.23 sql_string.h File Reference

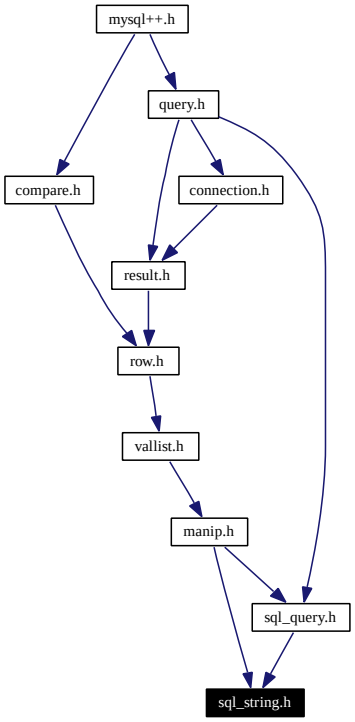
Declares an `std::string` derivative that adds some things needed within the library.

```
#include "defs.h"
#include <stdio.h>
#include <string>
```

Include dependency graph for sql_string.h:



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



Namespaces

- namespace **mysqlpp**

8.23.1 Detailed Description

Declares an `std::string` derivative that adds some things needed within the library.

This class adds some flags needed by other parts of MySQL++, and it adds conversion functions from any primitive type. This helps in inserting these primitive types into the database, because we need everything in string form to build SQL queries.

8.24 sqlplus.hh File Reference

Deprecated backwards-compatibility header. Use **mysql++.h** in new code instead.

8.24.1 Detailed Description

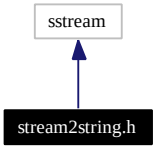
Deprecated backwards-compatibility header. Use **mysql++.h** in new code instead.

8.25 stream2string.h File Reference

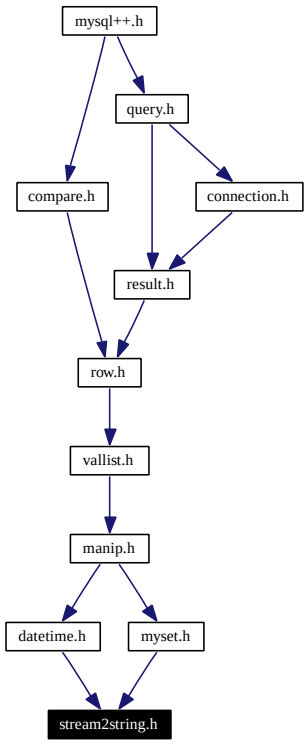
Declares an adapter that converts something that can be inserted into a C++ stream into a string type.

`#include <sstream>`

Include dependency graph for stream2string.h:



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



Namespaces

- namespace **mysqlpp**

8.25.1 Detailed Description

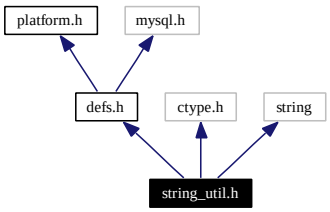
Declares an adapter that converts something that can be inserted into a C++ stream into a string type.

8.26 string_util.h File Reference

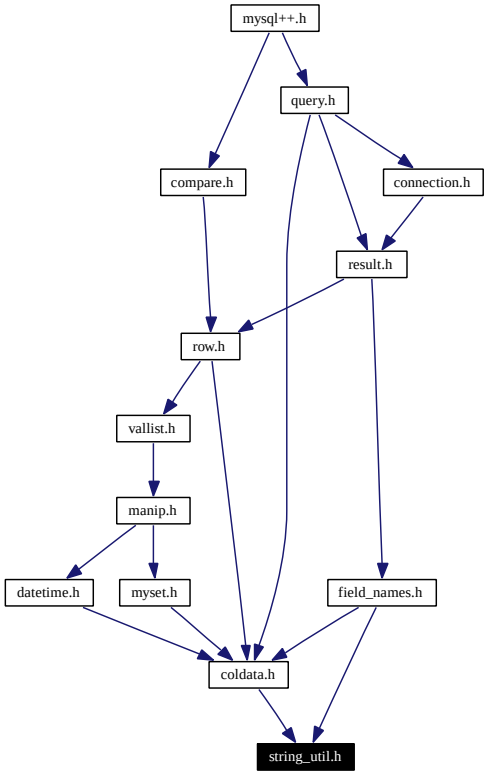
Declares string-handling utility functions used within the library.

```
#include "defs.h"
#include <ctype.h>
#include <string>
```

Include dependency graph for string_util.h:



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



Namespaces

- namespace **mysqlpp**

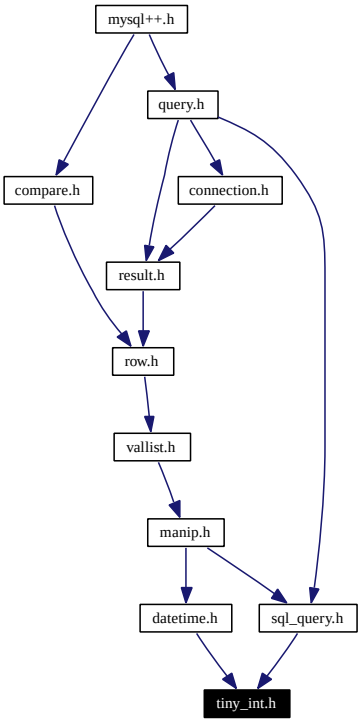
8.26.1 Detailed Description

Declares string-handling utility functions used within the library.

8.27 tiny_int.h File Reference

Declares class for holding a SQL tiny_int.

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



Namespaces

- namespace **mysqlpp**

8.27.1 Detailed Description

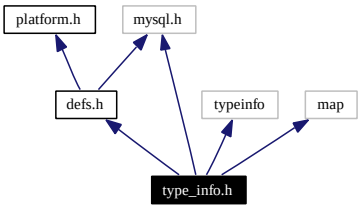
Declares class for holding a SQL tiny_int.

8.28 type_info.h File Reference

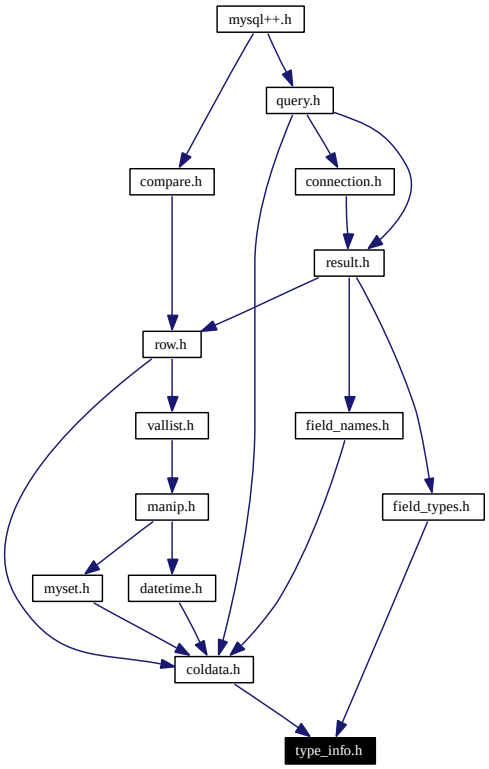
Declares classes that provide an interface between the SQL and C++ type systems.

```
#include "defs.h"
#include <mysql.h>
#include <typeinfo>
#include <map>
```

Include dependency graph for type_info.h:



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



Namespaces

- namespace **mysqlpp**

8.28.1 Detailed Description

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

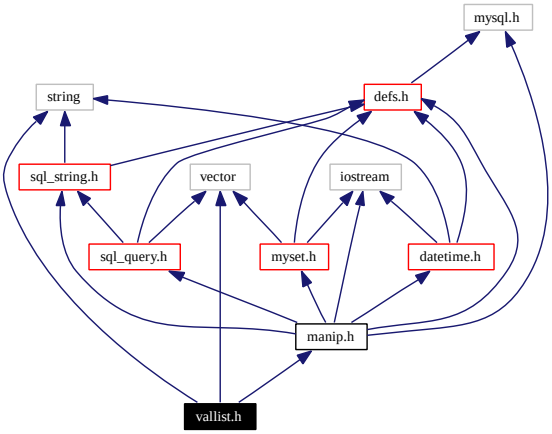
These classes are mostly used internal to the library.

8.29 vallist.h File Reference

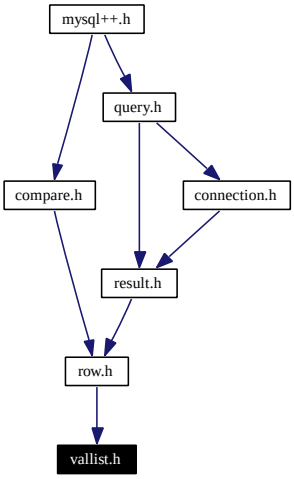
Declares templates for holding lists of values.

```
#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**

8.29.1 Detailed Description

Declares templates for holding lists of values.

Index