Document number:
Date: 2013-11-28
Project: Programming Language C++, Library Working Group
Reply-to: Johann Anhofer <johann.anhofer@gmail.com>
Status: Initial Draft

A Proposal to add a Database Access Layer to the Standard Library

I. Table of Contents

II. Introduction

This document describes an easy to use database abstraction layer for C++. It describes a set of classes which can be used to access an arbitrary database system.
There exist a lot of different database systems in the world, with a lot of different API's for accessing data within a C++ program. The most API's are in plain old C, which make the usage even harder for novice programmers. So we need a simple harmonized layer which can be easily extended for new database systems and also easily used by programmers.

III. Motivation and Scope

Accessing different database systems in a C++ program can be very cumbersome, because of the many different API's, almost all of them are in plain old C. There exists a number of abstractions to access different database systems with the same API. For example ODBC. But ODBC on it's own has a very large, verbose and complex structure, and it's also done in plain old C.

Different C++ Frameworks provide different abstraction layers for database access, which try to solve the problem described above. But they can't be used outside of the context of the frameworks. (E.g. Qt Framework, POCO Libraries, MFC, …).

This proposal tries to describe another abstraction layer, to reconcile all the different API's. The layer must be easy to understand/use and easy to extend for new database systems. It tries to extract the most common and needed features and sketch model classes reflecting this features.

1. Architecture

The general architecture for the database access layer consists of 4 sub layers.

1. Public Interface
2. Private interface
3. Database Client API
4. Database server/file

Only the public interface is visible to the user of the model classes. So there should be no need to struggle around with different API's.
The private interface describes objects specific to one database system. In the future this private layer should be delivered by the database vendors.
The private objects maybe only a layer for accessing the database client API, which itself communicates with the database file/server, but layer 2 and 3 can also be only one part.

The most important features to be implemented should be.

- transactions
- forwarding queries
- DDL statements
- named and positional parameters
- BLOB handling in chunks of data.
- Unicode support for column/parameter values.
- consequent null handling

IV. Impact on the Standard

The public and private interfaces (see part III) are provided by the standard. The implementation of the public interfaces are also provided by the standard. The implementation of the private interfaces are provided by database system vendors.

For transporting data from the users program to the database client API (Layer 3) there is a versatile data container, like the any class from the boost library, needed.

Some parts of the public interface use variadic templates and other C++ 11 features, like smart pointers. So C++ 11 is needed (at least for the sample implementation).

V. Design Decisions

Data transport

For data transport from and to the user code, a container class is used which is able to transport data of an arbitrary type (like boost::any), but not a variant based implementation like `QVariant` from QtSql [4]. Variants tend to influence the user code too much and `QVariant` has some odd behavior too (e.g. typed null values, `QVariant(QVariant::Int)` represents a null value of type integer, but null shouldn't have a type).

Null handling

So a separate class as null type will be introduced. Accessing the type or value of null always leads to an exception. Only checking for null with is_null( ) is valid.

Bindings

Also no binding to existing variables is introduced (like in [1], [2] and [3]), because this leads all to easy to dangling pointers/references. C++ 11 provides move semantics so return by value can be used without performance loss.

Iterators

No iterators are provided to read a result set, because a result set is not a container. It's time and memory consumption are not easily predictable (without looking at the query execution plan). Maybe iterators will lead novice users also to misuse standard library algorithms like std::sort, std::accumulate, … for which standard SQL functionality will be much better used, so the decision was not to provide iterators over result sets.

Database driver handling

For most flexibility on using different drivers for different database systems, a factory class for generating a driver object from a string key is introduced. So drivers which are loaded on runtime are able to register itself to this factory. This also prevents the user from handling with implementation details of a database driver.

VI. Technical Specifications

This section holds a detailed specification of the needed classes, helper classes and free functions. A sample implementation was done by the author of this proposal. Drivers for sqlite (a server less database engine [7]) and the open source database server firebird [6] are also provided by this implementation.

Classes of the public interface

The public interface consists of the following classes, this classes use driver specific implementation objects which implement the associated private interface.

Class	Description
driver_factory	A factory class, where database driver implementations can be registered, to be accessed via the connection object.
connection	The connection object establishes a connection to the specified database system.
statement	This class represents a SQL statement, which can be prepared, parameter values can be bound to it and then executed.
result	The access class for the result set of a SQL query.
transaction	This class handles database transactions.
parameters	An accessor class for the parameters of a SQL statement.

All these objects are moveable but not copyable. Except for driver_factory, which isn't movable too.

Now a detailed description of the classes follows.

1. driver_factory

The driver_factory is a singleton class, which provides functions for registering/deregistering database drivers via a driver name. A connection object can create a new instance of a registered driver via it's key. Each driver must implement the driver_interface (see Interfaces for objects of the private interface).

```
class driver_factory
{
public:
    static driver_factory &instance();

    using driver_creator_function = std::function<driver_interface *()>;

    void register_driver(const std::string &name, driver_creator_function creator);
    void unregister_driver(const std::string &name);

    std::shared_ptr<driver_interface> create_driver(const std::string &name) const;

    std::vector<std::string> registered_drivers() const;
};
```

The class contains a type **`driver_creator_function`** which is a callable to create an instance of a named driver.

instance	creates the instance on the first call and returns a reference to the one and only instance.
register_driver	Registers a callable for a driver name. The callable creates an instance of the driver object.
unregister_driver	Unregisters the driver.
`create_driver`	Creates an instance of the specified driver and returns a shared pointer to the driver_interface implementation.
registered_drivers	Returns an array of all registered driver names.

The driver_factory is typically used by driver implementers to register the driver and by the connection class to create an instance of the driver.

```
driver_factory::instance().register_driver(„sqlite“, []{return sqlite_driver::create();});
driver_factory::instance().unregister_driver(„sqlite“);

auto drv = driver_factory::instance().create_driver(„sqlite“);
```

Basically the driver object is used to create the private implementation objects for the public interface objects.

2. connection

The connection class establishes a connection to the database system. The user specifies the driver to use by it's name. The class uses the connection_interface to communicate with the private implementation of the connection.

```
class connection
{
public:
    explicit connection(const std::string &driver_name);

    using key_value_pair = std::unordered_map<std::string, std::string>;

    void open(const std::string &database, const authentication &auth =
              no_authentication{}, const key_value_pair & options = key_value_pair{});
    void close();
    bool is_open() const;
    handle get_handle() const;
};
```

On construction the user specifies the name of the database driver to use (which must be registered before to the driver_factory).

To establish/close a connection to the database system the `open/close` methods are used. The parameters for open are the database name, which may be a path or registered name, the authentication method which is discussed later (see Helper classes) and the options. The options are a list of key/value pairs, which describe properties specific to the used database driver. For example the `{{“encoding“, “UNICODE_FSS“}}` specifies the encoding character set for a connection to the firebird database.
The connection will be closed automatically if the destructor of the connection object is invoked. The user can manually close the connection with the close parameter.

Via `is_open` one is able to check if a connection object already holds an open connection or not.

The `get_handle` method retrieves a driver specific handle value to the native connection object of the used SQL API, if any.

Look at example 1 of section VIII to see how a connection to a sqlite in-memory database is established.

3. statement

The statement class handles preparation and execution of a SQL statement, as string. It also provides access to the parameters object, which is used to bind arguments to named and positional parameters.

```
class statement
{
public:
   statement(const std::string &sql_cmd, connection &conn);
   explicit statement(const connection &conn);

   void prepare(const std::string &sql_cmd);
   bool is_prepared() const;

   void reset();

   void execute_ddl();
   template<typename ...Args>
   void execute_ddl(Args&& ...args);
   void execute_ddl(std::initializer_list<parameter> params);

   void execute_non_query();
   template<typename ...Args>
   void execute_non_query(Args&& ...args);
   void execute_non_query(std::initializer_list<parameter> params);

   value execute_scalar();
   template<typename ...Args>
   value execute_scalar(Args&& ...args);
   value execute_scalar(std::initializer_list<parameter> params);

   result execute();
   template<typename ...Args>
   result execute(Args&& ...args);
   result execute(std::initializer_list<parameter> params);

   parameters get_parameters() const;

   handle get_handle() const;
};
```

On construction the user specifies the the connection object which is used to execute the statement. Optionally a SQL command string can be specified. In this case the SQL command will be prepared in the constructor body.

If no SQL command was specified at the constructor, a new SQL command can be prepared with the `prepare` function. The `is_prepared` function checks if a statement object already holds a prepared SQL statement.

After binding arguments to parameters and executing the statement a call to `reset` is used to reset the statement to the initial (prepared) state, so another set of arguments can be provided and the statement can be executed again (e.g. for consecutive inserts to the same table).
A SQL statement can be executed in four different ways.

1. If the SQL string holds a DDL statement, use `execute_ddl`
2. If the SQL string holds a DML statement which doesn't return rows, e.g. `INSERT INTO` or `DELETE FROM`, use `execute_non_query`.
3. If the SQL string holds a DML statement which returns only one row, use `execute_scalar` to retrieve the value of the first column of this row.
4. If the SQL string holds a DML statement which returns an arbitrary number of rows use `execute`.

For each of these four functions a set of three overloads are present.

1. With no parameters.
2. With an arbitrary number of parameters (as a variadic template)
3. With an initializer list of parameter objects (see data transport classes).

The third overload is the only way to use named parameters, the second overload is used for positional parameters.

The four execute functions also exists as free functions, to simplify the process of preparing a statement, binding arguments to the parameters and executing the statement. See „free functions“ for more informations.

The `get_parameters` function retrieves the parameters object, which is used to bind arguments to parameters of the SQL statement.

The `get_handle` method retrieves a driver specific handle value to the native statement object of the used SQL API, if any.

Look at example 2 of section VIII to see how a statement is used to execute SQL commands.

4. result

The `result` class represents the result set of a SQL query executed via the `statement` object.

```
class result
{
public:
   void move_next();
   bool is_eof() const;

   int get_column_count() const;

   std::string get_column_name(int column) const;
   int get_column_index(const std::string &column_name) const;

   bool is_column_null(int column) const;
   bool is_column_null(const std::string &column_name) const;

   value get_column_value(int column) const;
   value get_column_value(const std::string &column_name) const;

   value operator[](int column) const;
   value operator[](const std::string &column) const;

   handle get_handle() const;
};
```

The `result` class cannot be constructed by the user, the only way to get an instance of this class is to call the `execute` function of a `statement` object.

With subsequent calls to `move_next` all rows of a result-set can be processed from the first to the last. Only forward cursors are allowed for this, so no backward navigation is possible.

The `is_eof` returns true if no more rows are within the result set.

`get_column_count` returns the number of columns defined by the result-set.
Each column can be accessed via it's name or it's position. The name is retrieved by `get_column_name` and the position by `get_column_index`.

A column's value can be checked against null with the `is_column_null` functions (one overload for named columns, and one for column positions).

The `value` object for a column can be retrieved with the `get_column_value` functions (see „data transportation classes" for more information about the `value` class). The bracket operator is overloaded for convenience, it also returns the `value` object of a given column.

The `get_handle` method retrieves a driver specific handle value to the native result object of the used SQL API, if any.

Look at example 3 of section VIII to see how a result-set of a SQL query is used to retrieve data.

5. transaction

The `transaction` class is used to execute different SQL statement atomically.

```
class transaction
{
public:
   explicit transaction(const connection &conn);

   void begin();
   void commit();
   void rollback();

   bool is_open() const;

   handle get_handle() const;
};
```

A `transaction` object is created with the connection for which the transaction should be running. The destructor rolls the transaction back if neither `commit` nor `rollback` is called after calling begin. For automatic `begin/commit` sequences over a statement block use the `transaction_scope` class from „helper classes".

A transaction is started with a call to `begin`. If all goes well changes can be accepted via a call to `commit`, otherwise call `rollback` to reverse the changes. is_open checks if `begin` was called but not `commit` or `rollback`.

The get_handle method retrieves a driver specific handle value to the native transaction object of the used SQL API, if any.

Look at example 4 of section VIII to see a see how to use transactions.

6. parameters

The `parameter` class is used to bind arguments to parameters of a SQL statement.

```
class parameters
{
public:
   int get_count() const;
```

```
    template<typename T>
    void bind(int pos, T && value);
    template<typename T>
    void bind(const std::string &name, T && value);
    void bind(const parameter &param);

    handle get_handle() const;
};
```

An instance of a `parameter` class can only be retrieved by a call to the `get_parameters` function of the `statement` object.

`get_count` retrieves the number of parameters in the SQL statement.

`bind` is used to bind a value object to the given parameter of the associated SQL statement. The first overload is used for positional parameters. The second for named parameters and the third overload is provided to bind an instance of a parameter object (see „data transport classes“).

The get_handle method retrieves a driver specific handle value to the native parameters object of the used SQL API, if any.

Look at example 5 of section VIII to see how to bind arguments to SQL parameters.

Interfaces for objects of the private interface:

For each class in the public interface there exists a private interface, which provides access to the implementation of public objects. For each database system different implementations of this interfaces are needed. Such private implementation objects are created via the driver object which implements the driver_interface. All interfaces must have an empty virtual destructor, which isn't explicit stated in the description below.

1. driver_interface

This is the interface which all database driver objects have to implement.

```
struct driver_interface
{
  virtual connection_interface *make_connection() const = 0;
  virtual statement_interface *make_statement(const shared_connection_ptr &conn) const = 0;
  virtual parameters_interface *make_parameters(const shared_statement_ptr &stmt) const = 0;
  virtual result_interface *make_result(const shared_statement_ptr &stmt) const = 0;
  virtual transaction_interface *make_transaction(const shared_connection_ptr &conn) const = 0;
};
```

Each function of the driver object creates an instance of the private implementation of an object from the public interface. So for e.g. the `make_connection` function creates the private implementation for the connection class. The classes in the public interface mostly forward function calls to the implementation objects. Some objects need a second object to work, either `connection` or `statement`. So a shared pointer to the private implementation of these classes are forwarded to `make_statement`, `make_parameters`, `make_result` and `make_transaction`.

2. connection_interface

The most functions of the `connection_interface` are the same as described in the public interface.

```
struct connection_interface
{
virtual void open(const std::string &,
                  const authentication &auth = no_authentication{},
                  const key_value_pair & = key_value_pair{}
                  ) = 0;
virtual void close() = 0;
virtual bool is_open() const = 0;
```

```
virtual handle get_handle() const = 0;
virtual void set_current_transaction(const shared_transaction_ptr &trans) = 0;
virtual shared_transaction_ptr get_current_transaction() const = 0;
};
```

With `set_current_transaction` a transaction object is able to link a transaction to the connection object. On commit or rollback the transaction object set's the transaction to an invalid handle (e.g. nullptr).

Other objects can retrieve these transaction via `get_current_transaction`.

3. statement_interface

All functions of the `statement_interface` are part of the public interface of the `statement` class.

```
struct statement_interface
{
   virtual void prepare(const std::string &sqlcmd) = 0;
   virtual bool is_prepared() const = 0;
   virtual void execute_ddl() = 0;
   virtual void execute_non_query() = 0;
   virtual void execute() = 0;
   virtual void reset() = 0;
   virtual handle get_handle() const = 0;
};
```

4. result_interface

All functions of the `result_interface` are part of the public interface of the `result` class.

```
struct result_interface
{
   virtual int get_column_count() const = 0;
   virtual bool is_eof() const = 0;
   virtual void move_next() = 0;
   virtual value get_column_value(int column) const = 0;
   virtual value get_column_value(const std::string &column_name) const = 0;
   virtual std::string get_column_name(int column) const = 0;
   virtual int get_column_index(const std::string &column_name) const = 0;
   virtual bool is_column_null(int column) const = 0;
   virtual bool is_column_null(const std::string &column_name) const = 0;
   virtual handle get_handle() const = 0;
};
```

5. transaction_interface

All functions of the `transaction_interface` are part of the public interface of the `transaction` class.

```
struct transaction_interface
{
   virtual void begin() = 0;
   virtual void commit() = 0;
   virtual void rollback() = 0;
   virtual bool is_open() const = 0;
   virtual handle get_handle() const = 0;
};
```

6. parameters_interface

All functions of the `parameters_interface` are part of the public interface of the `parameters` class.

```
struct parameters_interface
{
   virtual int get_count() const = 0;
   virtual void bind(const parameter &) = 0;
   virtual handle get_handle() const = 0;
};
```

Helper classes

A few helper classes exist to support the public interface.

1. transaction_scope

This class issues a begin/commit transaction for a block scope. It starts the given transaction in the constructor body and commit's the transaction in the destructor body, if neither commit nor rollback was called in between.

```
transaction tr{conn}; // create a transaction for database connection conn.
{
  transaction_scope trs{&tr};
  // .. modify some data in the database
}
```

2. null_type

This special type is used to mimic database null values. Instances of this class are used to be passed as parameter value to the database and to retrieve null values from the database. Accessing a null-value leads to a `value_is_null` exception (see the `type_of` and `value_of` free functions). The only valid operation for a null value should be determining if it's a null value (see the `is_null` free function).

3. authentication, no_authentication, user_password_authentication

The authentication base class and two descendants, one for no authentication (e.g. for sqlite databases) and one for username/password authentication.

```
connection con_sqlite{“sqlite“};
con_sqlite.open(“:memory:“, no_authentication{});

connection con_firebird{“firebird“};
con_firebird.open(“localhost:employee“, user_password_authentication{“SYSDBA“, “masterkey“});
```

Data transport classes

This classes are used to transport values from and to the database.

4. value

A versatile type (like boost's any). It is able to carry a value of any datatype, but it's not a kind of a variant. So once it was created it isn't modifiable nor one can change the type of the transported value. The contents should be restricted, so that no reference types or pointer types could be stored, except for `const char *` and `const wchar_t *`.
value should be both copyable and movable. It's content are accessed via the free functions `type_of`, `value_of`, `is_null` and `cast_to`.

5. parameter

The parameter class is a pair of a name or index and an instance of a value class (see above). It is used to bind a parameter value to a named or positional parameter of a SQL command. It is copyable and movable.

The public interface is as following:

```
class parameter
{
public:
  parameter(int pos, value arg);
  bool has_index() const;
  int get_index() const;

  parameter(std::string name_in, value arg);
```

```
    bool has_name() const;
    std::string get_name() const;

    value get_value() const;
  };
```

The first constructor with the integer argument is for positional parameters. The pos argument is the number of the parameter placeholder in the SQL statement. 1 is the first parameter placeholder.
The second constructor is for named parameters.

With `has_index` a check for a positional parameter can be made and with `has_name` for a named parameter.

`get_index` retrieves the position of the parameter, if it is a positional parameter, otherwise an exception is thrown.

`get_name` retrieves the name of the parameter, if it is a named parameter, otherwise an exception is thrown.

`get_value` retrieves the argument value for the parameter.

Free functions

A number of free functions exists to simplify the usage of the public interface. First, the four execute variants of the `statement` class are also present as free functions. Each of them are overloaded for arbitrary number of parameter values of different types and an arbitrary number of `parameter` objects.

The second group of free functions are used to handle `value` and `parameter` types.

1. execute_ddl

```
template<typename ...Args>
void execute_ddl(connection &conn, const std::string &sql, Args && …args);

void execute_ddl(connection &conn, const std::string &sql,
                         std::initializer_list<parameter> params);
```

This function executes a DDL query `sql` against the database to which `conn` is connected. It takes an arbitrary number of arguments for the parameter placeholders in the SQL command. To use named parameters the second overload have to be used. See `statement::execute_ddl`.

2. execute_non_query

```
template<typename ...Args>
void execute_non_query(connection &conn, const std::string &sql, Args && …args);

void execute_non_query(connection &conn, const std::string &sql,
                       std::initializer_list<parameter> params);
```

This function executes a non querying SQL command `sql` against the database to which `conn` is connected. It takes an arbitrary number of arguments for the parameter placeholders in the SQL command. To use named parameters the second overload have to be used. See `statement::execute_non_query`.

3. execute_scalar

```
template<typename ...Args>
value execute_scalar(connection &conn, const std::string &sql, Args &&...args)

value execute_scalar(connection &conn, const std::string &sql,
                       std::initializer_list<parameter> params);
```

This function executes a SQL command `sql` against the database to which `conn` is connected and returns the value of the first column of the first row of the result set. It takes an arbitrary number of arguments for

the parameter placeholders in the SQL command. To use named parameters the second overload have to be used. See `statement::execute_scalar`.

4. execute

```
template<typename ...Args>
result execute(connection &conn, const std::string &sql, Args && ...args)

result execute(connection &conn, const std::string &sql,
                        std::initializer_list<parameter> params);
```

This function executes a SQL command `sql` against the database to which `conn` is connected and returns the result set. It takes an arbitrary number of arguments for the parameter placeholders in the SQL command. To use named parameters the second overload have to be used. See `statement::execute`.

5. type_of

```
template<typename T>
std::type_index type_of(const T &t);
```

This function determines the type of the argument. This is is used to get access to the type of the embedded type for a `value` or `parameter` object. If a `value` or `parameter` object holds the `null_type` a `value_is_null` exception is thrown. For all other `T`'s `type_of` returns `typeid(T)`.

6. value_of

```
template<typename T>
T value_of(const T &t);
```

This function is used to extract the embedded type for a `value` or `parameter` object. If `T` is the `null_type`, it throws a `value_is_null exception`. For all other `T`'s this function returns `t`. For `value` and `parameter` objects if `T` is not equal to the type of the embedded data or the embedded data is no convertible to `T`, a `type_mismatch` exception is thrown.

So, `value_of<long>(value{10})` returns `10l`, but `value_of<std::string>(value{10})` throws a `type_mismatch` exception. All integer types are convertible to each other and all floating point types are convertible to integer types and vice versa.

If a `value` or `parameter` object holds the `null_type`, `value_of` throws a `value_is_null` exception.

7. cast_to

```
template<typename T>
T cast_to(const value &val);

template<typename T>
T cast_to(const parameter &arg);
```

`cast_to` extends the possibilities of `value_of`. It is used to convert the contents of `value/parameter` to an arbitrary type, if this conversion is defined. So `value_of<std::string>(value{10})` should return `std::string("10")`.

If a `value` or `parameter` object holds the `null_type` a `value_is_null` exception is thrown.

8. is_null

```
template<typename T>
constexpr bool is_null(const T &t);
```

This function is used to check a type against the `null_type`. It returns true if `T` is the `null_type` or `T` is of type `value` or `parameter` with an embedded `null_type`. In all other cases `is_null` returns false.

Exceptions

For error handling, exceptions are used consequently.

- `value_is_null` derived from `std::logic_error`, is thrown if an instance of the `null_type` is accessed with `type_of`, `value_of` or `cast_to`.
- `type_mismatch` derived from `std::runtime_error` is thrown if the embedded data of a `value` or `parameter` object cannot be converted to the destination type using `value_of` or `cast_to`.
- `db_exception` derived from `std::runtime_error` is used as base class for more specific exceptions of a driver implementation. For instance `sqlite_exception` is used in the sample implementation of the sqlite driver.

VII. To Do

As with all work, this initial draft is far from complete. Much more work has to be invested to push it further, if the proposal get's accepted.
The database driver interface should be extended for a feature query facility. So a user of the library can determine, if a feature is available for the used database system, or not. Maybe string values can be used here. For instance `bool query_feature("named parameters")`. Maybe a more elaborate mechanism is needed.

BLOB handling for instance is completely missing, the sqlite sample implementation uses a `std::vector<uint8_t>` for smaller BLOB's which fit easily into memory. But handling BLOB's in chunks of data is crucial for larger BLOB's. So different database API's needs to be analyzed to extract the common parts and present it to the users of the library in an easy understandable manner.

Unicode handling is crucial for modern database usage. So we need a way to encode/decode Unicode strings from ANSI (char *) and wide characters (wchar_t *) to and from the database API's.

Now, the only support for authentication is plain-text username/password. A more sophisticated method is needed to pass credential data to the SQL client API's. Specially on windows, most database systems support integrated security access, which should be supported by the authentication classes. The type switching to determine which method was used in the sample implementation should be replaced by a more versatile solution.

It should be defined which conversions are allowed/implemented for `value_of` and `cast_to`.

The sample implementation uses the `tm` structure for date/time and timestamp values. It would be better to define separate date, time and timestamp classes to match the database data types. Time values up to nanoseconds are in use by database systems. Timestamp types also have to include timezones to match the need for cross-timezone timestamps.

VIII. Examples

1. Example #1: Connecting to a sqlite „in memory" database

```
#include "connection.h"
#include "execute.h"
#include "transaction.h"
#include "transaction_scope.h"

#include <iostream>

using namespace cpp_db;

int main()
{
   try
   {
      connection conn{"sqlite"};
      conn.open(":memory:");
```

```
      // execute sample code here
   }
   catch(const std::exception &ex)
   {
         std::cerr << “EXCEPTION: “ << ex.what() << std::endl;
   }
}
```

2. Example #2: Using the statement object and execute

```
   // use free function execute_ddl to create a table
   execute_ddl(conn, R“(create table test_table (
                        ID integer primary key,
                        NAME varchar(50),
                        AGE integer
                        );
                     )“);

   // insert 3 records
   statement stmt{R“(insert into test_table(ID, NAME, AGE, SALARY)
                     values(?, ?, ?, ?);)“, conn};
   stmt.execute_non_query(1, "Bilbo Baggins", 121, 1000.0);
   stmt.reset();
   stmt.execute_non_query(2, "Frodo Baggins", 33, 500.0);
   stmt.reset();
   stmt.execute_non_query(3, "Samwise Gamgee", 21, 250.0);

   std::cout   << "Cumulated salary: "
               << value_of<int>(
                     execute_scalar(conn, R“(select sum(SALARY)
                                             from test_table
                                             where AGE between ? and ?)“
                                    , 10, 100)
                  )
               << std::endl;
```

3. Example #3: Use the result object to extract data from a query

```
result res{execute(conn, "select * from test_table where AGE < ?", 100)};

// dump column names
for (int i = 0; i < res.get_column_count(); ++i)
   std::cout << res.get_column_name(i) << "\t";
std::cout << std::endl;

// dump data
while(!res.is_eof())
{
   std::cout   << cast_to<std::string>(res.get_column_value("ID")) << "\t"
               << value_of<std::string>(res.get_column_value(1)) << "\t"
               << value_of<int>(res["AGE"]) << "\t"
               << value_of<double>(res[3])
               << std::endl;

   res.move_next();
}
```

4. Example #4: Use transactions and transaction_scope

```
{
   transaction_scope trs{&trans};
   stmt.reset();
   stmt.execute_non_query(4, "Gandalf the grey", 9899, 5000.0);
}
```

5. Example #5: Bind parameters

```
statement stmt{R“(insert into test_table(ID, NAME, AGE, SALARY)
                  values(@ID, @Name, @Age, @Salary);)“, conn};
stmt.bind(parameter(“@ID“, 1));
```

```
stmt.bind(parameter(“@Name“, “Saruman the white“);
stmt.bind(parameter(“@Age“, 9909);
stmt.bind(parameter(“@Salary“, 10000.0);
stmt.execute();
```

IX. Acknowledgements

Many thanks go to Herb Sutter for his inflaming call to proposals in his Going Native 2013 talks.

X. References

[1] N3612 Desiderata of a C++11 Database Interface by Thomas Neumann
[2] N3458 Simple Database Integration in C++11 by Thomas Neumann
[3] N3415 A Database Access Library by Bill Seymour
[4] http://qt-project.org/doc/qt-5.1/qtdoc/topics-data-storage.html Data Storage QtDoc 5.1
[5] http://pocoproject.org/docs/00200-DataUserManual.html POCO Data User Guide
[6] http://www.firebirdsql.org Firebird opensource database
[7] http://sqlite.org sqlite serverless SQL database engine