N3458: Simple Database Integration in C++11

Thomas Neumann

Technische Univeristät München
neumann@in.tum.de

2012-10-22

Many applications make use of relational database to store and query their data. However, existing database interfaces like ODBC and JDBC introduce a significant barrier between the application logic and the database access itself. This makes database usage intimidating and cumbersome. Here, we propose a much simpler database integration mechanisms which makes use of C++11 features to allow for a very natural interaction with the database itself.

1 Motivation

While database access is very common for various kinds of application, existing database interfaces are quite cumbersome. For example consider the following somewhat simplified ODBC fragment (error handling omitted for readability)

```
SQLHandle statement;
SQLAllocHandle(SQL_HANDLE_STMT, connection,&statement);
SQLPrepare(statement,"select a,b from foo where c=? and d=?",38);
SQLLen len1=3;
SQLBindParameter(statement,1,SQL_PARAM_INPUT,SQL_C_CHAR,SQL_VARCHAR,len1,0,
    "foo",len1,&len1);
SQLINTEGER val=1234; SQLLen len2=sizeof(val);
SQLBindParameter(statement,2,SQL_PARAM_INPUT,SQL_C_SLONG,SQL_INTEGER,len2,0,
    &val,len2,&len2);
SQLExecute(statement);
while (SQLFetch(statement)==SQL_SUCCESS) {
   double a,b;
   SQLGetData(statement,1,SQL_C_DOUBLE,&a,0,0);
   SQLGetData(statement,2,SQL_C_DOUBLE,&b,0,0);
   cout << a << " " << b << endl;
}
```

Such a coding style is unfortunate for a number of reasons. First, a large amount of boilerplate code is required. Partly due to the nature of C (ODBC is a C binding), but also partly because the database interface is broken into a large number of function invocations. In addition, the whole code is basically dynamically typed and has a poor integration of the query result into the containing code.

In contrast to this, we propose a different API style that tightly integrates the database access into the containing C++ program. A motivating example is shown below (the code is semantically equivalent to the ODBC fragment above):

```
prepared_query<string,int> ps=conn.prepare_query("select a,b from foo
    where c=? and d=?");
double a,b;
for (auto row:ps("foo",1234).into(a,b))
   cout << a << " " << b << endl;
```

Note that the amount of code is much smaller, and would have been even smaller if we did not have used a prepared query. Note further, that both the query invocation and the query result integrate much more naturally in the C++ code. This greatly simplified the interaction with the database, and removes unnecessary usage barriers. In the simplest case a database query can look like

```
int a;
for (auto row:conn.query("select a from bar").into(a))
   cout << a << endl;
```

which is already quite close to the SQL statement itself. Going even further (e.g., automatically declaring "a") would be nice, but would require language changes, and we do not propose this here. We discuss possible syntax variations in Section 4.

In general, a database access layer should 1) offer a nice, and elegant interface to the database, and 2) should allow for efficient implementation. In this order. Simplifying query formulation and result retrieval is the foremost goal, but of course efficiency is also important. In this context it is important to note that query languages like SQL are declarative, and the user should be encouraged to use them in a declarative fashion. As such it does not appear to be a good idea to expose imperative implementation details like cursors to the user. Performing "result comprehension", i.e., looping over the result is a concept that is simple to understand and closely matches the internal implementation of most DBMSs. Therefore such an access pattern should be encouraged, the user can always buffer the query result himself if needed.

In the following we sketch a database layer that uses C++11 features to expose a more rich interface. This is intended to be a first step in discussing an appropriate interface. Nevertheless, the whole interface has been implemented and tested for a main-memory DBMS at the Technische Universität München.

The whole interface aims at exposing a strongly typed interface to the DBMS that allows for easy combination of C++ and database queries by using C++11 features. However, there are cases where the types are not known at compile time, for example with user-provided queries. Therefore, the access layer has *generic* classes, where types are not statically known, and strongly typed classes with known types. The generic classes can be converted (via move) into strongly typed classes, and this conversion

checks the involved types at runtime. As a result, the query interface is completely type-safe, and reasonable efficient, as this check is only done once per query.

2 Scope

The database interface presented here will usually be used for relational databases, and as such somewhat implies SQL (see ISO/IEC 9075:2011). However, it tries to be largely database agnostic due to practical reasons: Virtually no database system implements the complete SQL 2011 standard, there are unfortunately syntax variations for more rarely used features, also database systems evolve over time. The older SQL 92 standard is a useful guideline because most database system support it, but even that is only a hint. Therefore we only assume the following:

1. queries can be given to the database system in textual form. The interpretation of the text is up to the database system, the interface does not inspect it.

2. queries may contain placeholders for repeated invocations. The detection of these placeholders is up to the database system, the interface only allows for binding them with values.

3. after execution the query result is produced in tabular form, one row after the other.

Note that these assumptions do not really require SQL, one could offer such an interface for example in RDF databases, too. But most people will use SQL, and we will therefore consider our example query once again to illustrate the different points:

select a,b **from** foo **where** c=? **and** d=?

Semantically the query looks for all tuples in *foo* that have certain values for *c* and *d*. But for the database interface the semantic does not really matter, what is important is that 1) the query has two placeholder arguments (marked by "?"), and 2) the query produces two result columns (*a* and *b*). Placeholder arguments are used for efficiency reasons. Query compilation takes some time, and by using placeholders the same query can be executed over and over again with different bindings. For example some invocations might produce the following results:

c='a'∧d=3	
a	b
1	1
2	2

c='b'∧d=5	
a	b
3	2

From the perspective of the database interface a query is like a function invocation. It has zero or more parameters, and produces a table as result. The goal of this proposal is to make this query invocation as painless as possible, including simple access to the query result.

3 The Main Interface

The database connection itself is maintained and represented by a *connection* object, as show below.

```
class connection
{
   public:
   // Access modes
   enum class access_mode { read_only, read_write };
   // Unsigned data type for tuple counts
   typedef implementation defined cardinality_type;

   // Constructors
   connection();
   connection(const std::string& credentials,access_mode mode);
   connection(connection&& other);

   // Destructor
   ~connection();

   // No copies
   connection(const connection&) = delete;
   connection& operator=(const connection& other) = delete;

   // Move assignment
   connection& operator=(connection&& other);

   // Connect to the server
   void connect(const std::string& credentials,access_mode mode);
   // Close the connection
   void close();

   // Prepare a query
   generic_prepared_query prepare_query(const std::string& query);
   // Prepare a statement
   generic_prepared_statement prepare_statement(const std::string&
       statement);

   // Execute an (un-prepared) query
   template <typename... parameter_types>
   generic_query_result query(const std::string& query,parameter_types
       const& ... arguments)
   { return prepare_query(query)(arguments); }
   // Execute an (un-prepeared) statement
   template <typename... parameter_types>
   cardinality_type execute(const std::string& statement,parameter_types
       const& ... arguments)
   { return prepare_statement(statement)(arguments); }
};
```

It primarily holds the connection, and allows for issuing queries. As a simple query can be implemented by implicitly first preparing the query and then executing it, we concentrate on the prepared queries part for now. The *prepare_query* method gets the

query as a string, and returns a *generic_prepared_query* object. The *generic* part in the name implies that we do not statically know the types of the query parameters (if any). As we will see in a moment we can add that information if available. For now we consider the generic class, as shown below:

```
class generic_prepared_query
{
   public:
   // Constructors
   generic_prepared_query();
   generic_prepared_query(generic_prepared_query&& other);
   // Destructor
   ~generic_prepared_query();

   // Assignment
   generic_prepared_query& operator=(generic_prepared_query&& other);

   // No copies
   generic_prepared_query(const generic_prepared_query&) = delete;
   generic_prepared_query& operator=(const generic_prepared_query& other)
      = delete;

   // Examine the query parameters
   unsigned parameter_count() const;
   type parameter_type(unsigned index) const;

   // Bind the next parameters
   template <class T> void bind(const T& value);
   void bind(nullptr_t null);

   // Run the query
   generic_query_result operator()();
};
```

The class allows for examining the query parameters (if any), and for binding them in sequential order. Overloading is used to detect binding NULLs (nullptr). Finally, the operator () executes the query, but we ignore this for a moment. For now we consider the more interesting case of prepared queries, namely prepared queries with statically typed parameters. They are defined below

```
template <typename... parameter_types> class prepared_query
{
   public:
   // Constructor
   prepared_query();
   prepared_query(prepared_query&& other);
   // Destructor
   ~prepared_query() {}

   // Assignment
   prepared_query& operator=(prepared_query&& other);

   // No copies
   prepared_query(const prepared_query&) = delete;
```

```
    prepared_query& operator=(const prepared_query& other) = delete;

    // Converted a generic-typed prepared query into a statically typed one
    prepared_query(generic_prepared_query&& other);
    prepared_query& operator=(generic_prepared_query&& other);

    // Run the query
    generic_query_result operator()(parameter_types const& ... arguments);
};
```

Note that a generic prepared query can be converted into a strongly typed query using the appropriate move constructor or move assignment. This conversion method checks the types involved at runtime, and reports an error in the case of mismatches. Afterwards, the query can be executed by simply invoking the object with the appropriate query parameters.

The query result itself is represented by the a query result object, again in a generic flavor for dynamic types and a strongly typed flavor. All query invocations return the generic object, which can then be converted into the properly typed flavor. The definition of the generic version is

```
class generic_query_result
{
    public:
    // Unsigned data type to reference columns
    typedef implementation defined index_type;

    // Constructor
    generic_query_result();
    generic_query_result(generic_query_result&& other);
    // Destructor
    ~generic_query_result();

    /// Assignment
    generic_query_result& operator=(generic_query_result&& other);

    // No copies
    generic_query_result(const generic_query_result&) = delete;
    generic_query_result& operator=(const generic_query_result& other) =
        delete;

    // Inspect the column schema
    index_type column_count() const;
    std::string column_name(index_type index) const;
    type column_type(index_type index) const;

    // Row access
    class row {
        public:
        // Constructor
        row();
        row(row&& other);

        // Assignment
```

```
        row& operator=(row&& other);

        // No copies
        row(const row&) = delete;
        row& operator=(const row&) = delete;

        // Is the column NULL?
        bool null(index_type index) const;
        // Access a column
        template <class T> T column(index_type index) const;
        // Get a textual representation of the column
        std::string to_string(index_type index) const;
    };
    /// Row access, Input-iterator
    class row_iterator {
        public:
        // Constructor
        row_iterator();
        row_iterator(const row& other);
        row_iterator(row_iterator&& other);

        // Assignment
        row_iterator& operator=(const row& other)
        row_iterator& operator=(row_iterator&& other);

        // Comparison
        bool operator==(const row& other) const;
        bool operator!=(const row& other) const;

        // Go to the next tuple
        row_iterator& operator++();
        // Access
        row operator*();
    };
    // Access the tuples
    row_iterator begin();
    row_iterator end();

    // Bind columns and transfer result into other object
    template <typename... column_types> query_result<column_types...>
        into(column_types&... host_variables);
};
```

The generic query result allows for iterating over the result (via Input-iterator, i.e., only once!), and each tuple can be inspected using access methods. Again, the more interesting case is the conversion into a strongly typed query result, that is done by invoking the *into* method. The method arguments both provide the target variables where the column values should be stored, and imply the data types of the underlying column. The strongly typed class is defined as

```
template <typename... column_types> class query_result
{
    public:
    // Constructor
```

```
    query_result();
    query_result(query_result&& other);
    // Destructor
    ~query_result();

    // Assignment
    query_result& operator=(query_result&& other);

    // No copies
    query_result(const query_result&) = delete;
    query_result& operator=(const query_result& other) = delete;

    // A row. Note that the real result is passed as a side effect!
    class row {
    };
    // Row access, Input-iterator
    class row_iterator {
       public:
       /// Constructor
       row_iterator();
       row_iterator(const row_iterator& other);
       row_iterator(row_iterator&& other);

       // Assignment
       row_iterator& operator=(const row_iterator& other);
       row_iterator& operator=(row_iterator&& other);

       // Comparison
       bool operator==(const row_iterator& other) const;
       bool operator!=(const row_iterator& other) const;

       // Increment
       row_iterator& operator++();
       /// Access the row
       row operator*();
    };
    // Access the result
    row_iterator begin();
    row_iterator end();
};
```

Again, the class allows for iterating over the result using an Input-iterator. That is, it is designed to be used in a range based for loop, which allows for a very convenient syntax. What is a bit surprising is that dereferencing the row iterator does not give the result tuple. Or rather, it does, but it appears as a side effect in the host variables. When looking at the individual operations that seems to be a surprising behavior, but when using the query result with a range-based for loop, as it was intended, this behavior is invisible to the user. On the plus side, the query result is directly available to the user without fiddling around with *tuple* objects, which is very convenient.

All queries and statements are by default executed in an "auto-commit" mode, i.e., each is considered as a separate transaction. Multiple queries and statements can be aggregated into a larger transaction by instantiating a transaction object

```
class transaction
{
   public:
   // Constructor
   explicit transaction(connection& conn);
   transaction(transaction&& other);
   // Destructor
   ~transaction();

   // Assignment
   transaction& operator=(transaction&& other);

   // No copies
   transaction(const transaction&) = delete;
   transaction& operator=(const transaction& other) = delete;

   // Commit the transaction
   void commit();
   // Rollback the transaction explicitly. This operation always succeeds.
   void rollback() noexcept;
};
```

The constructor of the transaction begins a database transaction, and the destructor implicitly triggers a rollback unless *commit* (or an explicit *rollback*) was invoked before. It is an error to invoke queries or statements between invoking commit/rollback and executing the destructor (i.e., a commit/rollback must be the last operation of a transaction). Note that rollbacks by definition always succeed, therefore a rollback in the destructor is safe regarding exceptions.

Some more classes are required for a full database interface, in particular for statements, but as they are quite similar to the query case we disregard them in this first proposal. They can be defined analogously.

4 Syntax Variations

The foremost goal of this proposal is to allow for a simple, easy to use integration of database access into C++ code. The integration is not perfect because C++ and SQL are quite different, and furthermore the SQL schema is only known at runtime. Therefore we have to accept some compromises. As shown above, the currently proposed syntax for a query is:

```
double a,b;
for (auto row:query("foo",1234).into(a,b))
   cout << a << " " << b << endl;
```

This syntax is compact and readable, but has two slight oddities: First, the *row* variable is never used. It exists only due to syntax requirements of C++11. Arguably one could extend *auto* to allow for unnamed variables, similar to unnamed function parameters, which would then lead to the syntax

```
double a,b;
```

```
for (auto:query("foo",1234).into(a,b))
   cout << a << " " << b << endl;
```

This avoids the unnecessary variable, and is quite readable. The second oddity that exists in both variants is that the result variables (a and b) are filled as a side effect of the iteration. This is not bad per se, and in fact for database users the *into* syntax is well known from embedded SQL, so this syntax is fine, but it is different from other standard classes. The "traditional" C++11 syntax would be

```
for (auto row:query<double,double>("foo",1234))
   cout << get<0>(row) << " " << get<1>(row) << endl;
```

Here, the query returns a regular C++11 tuple which can then be examined. But while this fits most naturally with the existing practice, it is actually a poor choice for query integration: First the `get<N>(row)` is quite verbose, much longer then a simple `a`. Note that the for loop might be much larger than our simple example, and it might reference columns multiple times. Second, the column numbers are hard to remember, and, even worse, they change if the *select* part of the query is modified. This makes updating queries quite painful.

A variant that would be possible with some limited compile-time reflection (see N3403) would be

```
struct Row { double a,b; };
for (auto row:query<Row>("foo",1234))
   cout << row.a << " " << row.b << endl;
```

but that would require language changes. In general it would be useful to be able to assign names to regular C++11 tuple entries, for example like this

```
for (auto [a,b]:query<double,double>("foo",1234))
   cout << a << " " << b << endl;
```

but as this would require more drastical language changes we are not proposing this. Given the current language restrictions, we think that the syntax proposed first is probably the best choice, and it would become even more pleasant to use if unnamed auto variables were allowed.

5 Type System

A problem we have side-stepped so far is the problem of SQL data types. SQL defines several data types that are not directly available inside standard C++. First, all data types exist in NULL-able variants, which are not available in C++. NULL-able types can be handled reasonably well using a template wrapper. In particular the *optional* proposal from N3406 can be used to model NULL-able types in C++. From a database perspective it might be nicer to call the wrapper *nullable* instead of optional, but that can be achieved easily by using

```
template <typename T> using nullable = optional<T>;
```

We will therefore ignore the problem of NULLs and concentrate on non-NULL types. But even there, many types do not exist in C++. For example fixed point numbers

are quite essential to SQL, but not available to C++. Floating point values like *double* are a poor approximation here, as the conversion introduces errors, which might not be acceptable in some application domains. It would therefore seem to be prudent to explicitly define a corresponding data type in C++ for every SQL 92 data type. There are not too many of these, and most of the definitions are simple. Even fix-point numbers are simple if we only want to store them (and optionally convert/print them) and disregard arithmetic.

Another issue that warrants some thoughts are BLOB values, i.e., very large strings or byte arrays. Databases usually handle them separately for implementation reasons, and ODBC for example has specific methods to retrieve and insert them. This could be solved by adding a BLOB-container type that lazily handles BLOB access. On the other hand C++11 with move semantics allows for efficient passing of large values, so even a std::string might be good enough for BLOBs. And using std::string would definitively improve usability, as then BLOBs were an implementation detail that does not affect the user.

This first draft disregards the typing problems for now, but this will have to be addressed in further revisions.

6 Conclusion

Database bindings should offer a simple and elegant interface to the database. Here, we propose a database interface that is both easy to use and flexible. Internally, the binding has to differentiate several different scenarios, as type information might or might not be available statically. But to the user this complexity is largely opaque, as generic types can be transparently converted into strongly typed variants, which offer a much simpler interface.

Acknowledgements: Thanks to Jens Maurer for his help in preparing this document and many insightful comments.