Selected C++11 Template Toffees from sqlpp11

Dr. Roland Bock

2014-02-27

String-based SQL

Creating tables

Creating tables

Who can read that?

```
CREATE TABLE tab_person
(
    id int AUTO_INCREMENT PRIMARY KEY,
    name varchar(255) NOT NULL,
    quirk int NOT NULL
);

CREATE TABLE tab_quirk
(
    id int AUTO_INCREMENT PRIMARY KEY,
    name varchar(255) NULL DEFAULT "",
    fatal bool NOT NULL
);
```

Some sample SQL statements

Some sample SQL statements

What's wrong here?

```
INSERT INTO Customs (CustomerName, ContactName,
                     Address, City, PostalCode, Country)
VALUES ('Cardinal','Tom B. Erichsen',
        'Stavanger','Skagen 21',4006,'Norway');
SELECT * FROM Orders,
INNER JOIN Customers
ON Orders.ConsumerID==Customers.CustomerIP;
```

(Copied with slight modifications from http://www.w3schools.com/sql)

What's wrong with text based SQL?

What's wrong with text based SQL?

It is not too bad, actually.

- SQL statements are vendor specific
- But you get immediate feedback in many cases when you are doing something wrong!

What's wrong with string-based SQL libraries in C++?

What's wrong with string-based SQL libraries in C++?

Example from http://cppcms.com/sql/cppdb/intro.html

```
res = sql << "SELECT n,f FROM test WHERE id=?" << 1 << cppdb::row;
if(!res.empty())
{
        int n = res.get<int>("n");
        double f=res.get<double>(1);
}
```

What's wrong with string-based SQL libraries in C++?

When we write the code

- We know the exact structure of the statements
- We know which columns belong to which tables
- We know the types of each column
- We know some additional semantics like which columns have automatic values or are supposed to be read-only, etc

What's wrong with string-based SQL libraries in C++?

When we write the code

- We know the exact structure of the statements
- We know which columns belong to which tables
- We know the types of each column
- We know some additional semantics like which columns have automatic values or are supposed to be read-only, etc

And then?

We write it into a flat string with no structure, no types, nothing.

What's wrong with string-based SQL libraries in C++?

Thus, all the useful information is thrown away while writing the code.

What's wrong with string-based SQL libraries in C++?

Thus, all the useful information is thrown away while writing the code.

The immediate feedback is gone!

All the possible bugs lie in wait until runtime.

What's wrong with string-based SQL libraries in C++?

Thus, all the useful information is thrown away while writing the code.

The immediate feedback is gone!

All the possible bugs lie in wait until runtime.

And our strings are vendor specific which makes our code vendor specific.

String-based SQL

Why do we use text-based SQL libraries in C++?

String-based SQL

Why do we use text-based SQL libraries in C++?
I think we shouldn't. There are alternatives.

sqlpp11

sqlpp11

sqlpp11 teaches SQL to the C++ compiler

sqlpp11

sqlpp11 teaches SQL to the C++ compiler

What does that mean?

- You can tell your compiler about your databases, tables and columns
- You can express the structure of your statements
- You will get structured, well defined results

sqlpp11

sqlpp11 teaches SQL to the C++ compiler

What does that mean?

- You can tell your compiler about your databases, tables and columns
- You can express the structure of your statements
- You will get structured, well defined results

And of course:

- You will get immediate feedback when doing something wrong!

  (In most cases, at least)

sqlpp11

sqlpp11 teaches SQL to the C++ compiler

What does that mean?

- You can tell your compiler about your databases, tables and columns
- You can express the structure of your statements
- You will get structured, well defined results

And of course:

- You will get immediate feedback when doing something wrong!

  (In most cases, at least)

And your expressions are vendor neutral (unless you *need* vendor-specific things).

sqlpp11

sqlpp11

Vendor neutral expressions.

connector

Translates between sqlpp11 and the database

Database

Vendor specific.

sqlpp11

Let's see some expressions.

Select examples

Select some

```
int main()
{
  for (const auto& row : db(select(p.name, p.quirk)
                            .from(p)
                            .where(p.id > 17
                              and p.name != "Zed")))
  {
    std::string name = row.name;
    int64_t quirk = row.quirk;
  }
}
```

Select examples

Use an alias

```
SQLPP_ALIAS_PROVIDER(left);
SQLPP_ALIAS_PROVIDER(right);
int main()
{
  for (const auto& row : db(
               select(p.name.as(left), q.name.as(right))
              .from(p.join(q).on(p.quirk == q.id))
              .where(p.id > 29)))
  {
    std::string person_name = row.left;
    std::string quirk_name = row.right;
  }
}
```

Select examples

Select something complex

```
int main()
{
  auto s = select(all_of(p))
              .from(p)
              .where(p.name == any(select(q.name)
                                   .from(q)
                                   .where(true)))
              .group_by(q.name)
              .having(p.name.like("%Bee%"))
              .order_by(p.name.asc())
              .limit(3).offset(7);
}
```

What if I do something wrong?

What if I do something wrong?

Bad insert

```
int main()
{
  insert_into(p).set(p.quirk = "template error");
}
```

What if I do something wrong?

Bad insert

```
int main()
{
  insert_into(p).set(p.quirk = "template error");
}
```

Errors

```
In file included from /temp/Sample.h:4:
In file included from /home/rbock/projects/techtalks/include/sqlpp11/table.h:32:
/include/sqlpp11/column.h:92:5: error: static_assert failed "invalid assignment operand"
              static_assert(sqlpp::vendor::wrong_t<T>::value, "invalid assignment operand");
              ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/temp/bad-insert.cpp:37:29: note: in instantiation of function template specialization
      'sqlpp::column_t<test::TabPerson, test::TabPerson_::Quirk>::operator=<const char *>'
       requested here insert_into(p).set(p.quirk = "template error");
                                                 ^
1 error generated.
```

What if I do something wrong?

Bad select

```
int main()
{
  for (const auto& row : db(
              select(all_of(p).as(p),
                  all_of(q).as(p.quirk))
              .from(p.join(q).on(p.quirk == q.id))
              .where(p.id > 29 and q.fatal == "yes")))
  {
    int64_t person_id = row.tabPerson.id;
    std::string quirk_name = row.quirk.name;
  }
}
```

What if I do something wrong?

```
In file included from /temp/Sample.h:4:
In file included from /include/sqlpp11/table.h:30:
/include/sqlpp11/type_traits.h:137:4: error: static_assert failed "invalid operand type"
                        static_assert(IsCorrectType<type>::value, "invalid operand type");
                        ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~
/include/sqlpp11/basic_operators.h:49:36: note: in instantiation of template class
      'sqlpp::operand_t<const char *, is_boolean_t>' requested here
                              vendor::equal_t<Base, typename Constraint<T>::type> operator==(T t) const
                                                                  ^
/include/sqlpp11/basic_operators.h:49:57: note: while substituting deduced template arguments into function te
      'operator==' [with T = const char *]
                              vendor::equal_t<Base, typename Constraint<T>::type> operator==(T t) const
                                                                                  ^
/temp/bad-select.cpp:41:38: warning: result of comparison against a string literal is unspecified (use strncmp
      [-Wstring-compare]
                                                 .where(p.id > 29 and q.fatal == "yes")))
                                                                                 ^  ~~~~~
/temp/bad-select.cpp:41:38: error: invalid operands to binary expression ('sqlpp::column_t<test::TabQuirk,
      test::TabQuirk_::Fatal>' and 'const char *')
                                                 .where(p.id > 29 and q.fatal == "yes")))
                                                                        ~~~~~~~ ^  ~~~~~
/include/sqlpp11/basic_operators.h:49:57: note: candidate template ignored: substitution failure [with T = con
                              vendor::equal_t<Base, typename Constraint<T>::type> operator==(T t) const
                                                                                  ^
1 warning and 2 errors generated.
```

How does this work?

Let's get a taste of the library's code.

Template Toffees

Sweet and delicious template patterns. . .

Template Toffees

Sweet and delicious template patterns...
(An acquired taste, of course)

Member Names And Types

Selected example

```
SQLPP_ALIAS_PROVIDER(feature);

for (const auto& row : db(select(p.name,
                                 p.quirk.as(feature))
                          .from(p)
                          .where(p.id > 17 and p.name != "Zed")))
{
  std::string name = row.name;
  int quirk = row.feature;
}
```

Member Name Toffee

Member Name Toffee

Member Name Template

```
template<typename T>
struct _member_t
{
  T quirk;
};
```

Member Name Toffee

Member Name Template

```
template<typename T>
struct _member_t
{
  T quirk;
};
```

Basic Usage

```
struct my_t: public _member_t<int>
{
};
```

A real world column definition

```
struct Id
{
  struct _name_t
  {
    static constexpr const char* _get_name() { return "id"; }
    template<typename T>
    struct _member_t
    {
      T id;
      T& operator()() { return id; }
      const T& operator()() const { return id; }
    };
  };
  using _value_type = sqlpp::integer;
  struct _column_type {};
};
```

A real world table definition

```
struct TabPerson: sqlpp::table_t<TabPerson, Id, Name, Quirk>
{
  struct _name_t
  {
    static constexpr const char* _get_name() { return "tab_person"; }

    template<typename T>
    struct _member_t
    {
      T tabPerson;
      // operators
    };
  };
  using _value_type = sqlpp::no_value_t;
};
```

The table template

```
template<typename Table, typename... ColumnSpecs>
struct table_t:
   public ColumnSpecs
          ::_name_t
          ::template _member_t<column_t<Table, ColumnSpecs>>...
{
  // ...
};
```

Named Parameters

Named Parameters

Insert

```
int main()
{
  db(insert_into(q).set(q.name = "loves c++",
                        q.fatal = false));
}
```

Named Parameter Toffee

Assignment

```
template<typename Lhs, typename Rhs>
struct assignment_t
{
  using lhs_t = Lhs;
  using rhs_t = Rhs;


  lhs_t _lhs;
  rhs_t _rhs;
};
```

Named Parameter Toffee

Assignment

```
template<typename Lhs, typename Rhs>
struct assignment_t
{
  using lhs_t = Lhs;
  using rhs_t = Rhs;


  lhs_t _lhs;
  rhs_t _rhs;
};
```

Named Parameter

```
struct first_name_t
{
  auto operator=(std::string arg)
    -> assignment_t<first_name_t,
                    std::string>
  {
    return { {}, arg };
  }
};
constexpr first_name_t first_name{};
```

Usage In Expression Templates

Usage In Expression Templates

Insert

```
template<typename... Assignments>
struct insert_list_t
{
  // check arguments here

  insert_list_t(Assignments... assignments):
    _columns(assignments._lhs...),
    _values(assignments._rhs...)
    {}

  std::tuple<typename Assignments::_lhs_t...> _columns;
  std::tuple<typename Assignments::_rhs_t...> _values;
};
```

Variadic Type Checks

How do we check arguments?

Examples from sqlpp11

Select

```
test::TabPerson p;
select(p.name, p.quirk)
    .from(p, q)
    .where(p.id > 17
        and p.name != "Zed");
```

Examples from sqlpp11

Select

```
test::TabPerson p;
select(p.name, p.quirk)
    .from(p, q)
    .where(p.id > 17
        and p.name != "Zed");
```

- Columns have to be named expressions
- Column names have to be unique
- Columns have to match the tables in FROM
- Table names in FROM have to be unique
- WHERE-expression has to be a bool expression
- WHERE-expression must use tables from FROM

And all checks should be performed at compile time.

Type checking

So we have these kinds of checks:

- All arguments must meet a certain requirement
- Types must have to be unique
- Types have to be elements of a set of types.

A real life example

Insert assignments

```
template<typename... Assignments>
struct insert_list_t
{
  static_assert(sizeof...(Assignments),
                "at least one select expression required in set()");

  static_assert(not ::sqlpp::detail::has_duplicates<Assignments...>::value,
                "at least one duplicate argument detected in set()");

  static_assert(sqlpp::detail::and_t<is_assignment_t, Assignments...>::value,
                "at least one argument is not an assignment in set()");

  static_assert(not sqlpp::detail::or_t<must_not_insert_t,
                                        typename Assignments::_column_t...>::value,
                "at least one assignment is prohibited by its column definition in

};
```

Invalid template arguments

Tuple argument expected

```
template<typename T>
struct recipe
{
  static_assert(???????, "invalid argument, tuple expected");
};

template<typename... Ingredients>
struct recipe<std::tuple<Ingredients...>>
{
  //...
};
```

The Wrong Toffee

The Wrong Toffee

This is just wrong

```
template<typename... T>
struct wrong_t
{
  static constexpr bool value = false;
};
```

The Wrong Toffee

This is just wrong

```
template<typename... T>
struct wrong_t
{
  static constexpr bool value = false;
};
```

This is much worse, though

```
template<typename... T>
using worse_t = std::false_type;
```

Invalid template arguments

Tuple argument expected

```
template<typename T>
struct recipe
{
  static_assert(wrong_t<T>::value, "invalid argument, tuple expected");
};

template<typename... Ingredients>
struct recipe<std::tuple<Ingredients...>>
{
  //...
};
```

Partial Specialization Of Functions

Partial Specialization Of Functions

There is no partial specialization for functions, but what if you wanted to have it?

Partial Specialization Of Functions

String concatenation requires specialization

```
template<typename Left, typename Right>
struct concat_t
{
  template<typename Context>
  void serialize(Context& context) const
  {
    _left.serialize(context);
    context << "||";
    _right.serialize(context);
  }
  //...
```

For MySQL we need a different version, using the CONCAT method.

Partial Specialization Of Functions

String concatenation requires specialization

```
template<typename Left, typename Right>
struct concat_t
{
  template<typename Context>
  void serialize(Context& context) const;
  //...
};
```

Partial Specialization Of Functions

String concatenation requires specialization

```
template<typename Left, typename Right>
struct concat_t
{
  template<typename Context>
  void serialize(Context& context) const;
  //...
};
```

We require partial specialization because there are three parameters (Left, Right and Context), but we want to specialize only one of them: Context

Partial Specialization Of Functions

What are the options?

Partial Specialization Of Functions

What are the options? There are several ways to do it, e.g.

- Overloading, but that would require us to know the Context classes, and there could be hundreds.
- Using static tags or flags in the Context class, to differentiate between the options. But again there could be hundreds.
- Dispatch to a static method of a template struct or class and use partial specialization.

We'll go with the last option because it enables the author of the Context class to handle the specialization where required.

Dispatch To Static Method Of A Template Struct

Dispatch from member method

```
template<typename Left, typename Right>
struct concat_t
{
  template<typename Context>
  void serialize(Context& context) const
  {
    serializer_t<Context, concat_t>::_(*this, context);
  }
};
```

Dispatch To Static Method Of A Template Struct

Dispatch from member method

```
template<typename Left, typename Right>
struct concat_t
{
  template<typename Context>
  void serialize(Context& context) const
  {
    serializer_t<Context, concat_t>::_(*this, context);
  }
};
```

But since this now is a generic method, we can also use a very simple free function.

Dispatch To Static Method Of A Template Struct

Dispatch from free function

```
template<typename Context, typename T>
struct serializer_t
{
  static void _(const T& t, Context& context)
  {
    static_assert(wrong_t<Context, T>::value, "missing serializer specialization");
  }
};

template<typename T, typename Context>
void serialize(const T& t, Context& context)
{
  serializer_t<Context, T>::_(t, context);
}
```

Dispatch To Static Method Of A Template Struct

MySQL's concat

```
template<typename Left, typename Right>
struct serializer_t<mysql::context_t, concat_t<Left, Right>>
{
  using T = concat_t<Left, Right>;
  static void _(const T& t, mysql::context_t& context)
  {
    context << "CONCAT(";
    serialize(t._left, context);
    context << ',';
    serialize(t._right, context);
    context << ')';
  }
};
```

The Expression Tree

We got a glimpse of how to serialize it.
What else could you do with it?

Names can be deceiving

Renamed serializer

```
template<typename Context, typename T>
struct interpreter_t
{
  static void _(const T& t, Context& context)
  {
    static_assert(wrong_t<Context, T>::value, "missing interpreter special
  }
};

template<typename T, typename Context>
void interpret(const T& t, Context& context)
{
  vendor::interpreter_t<Context, T>::_(t, context);
}
```

And Now For Something Completely Different

Another Assignment

```
namespace vendor
{
  template<typename Lhs, typename Rhs>
  struct interpreter_t<container::context_t,
                       assignment_t<Lhs, Rhs>>
  {
    using T = assignment_t<Lhs, Rhs>;

    static auto _(const T& t, container::context_t& context)
      -> container::assignment_t<decltype(interpret(t._lhs, context)),
                                 decltype(interpret(t._rhs, context))>
    {
      return { interpret(t._lhs, context), interpret(t._rhs, context) };
    }
  };
}
```

Reinterpreted Assignment

Looks familiar?

```
namespace container
{
  template<typename Lhs, typename Rhs>
  struct assignment_t
  {
    template<typename T>
    void operator()(T& t)
    {
      _lhs(t) = _rhs(t);
    }
    Lhs _lhs;
    Rhs _rhs;
  };
}
```

Reinterpreted Assignment

This reinterprets the compile time expression tree into a tree of functors that can be called at runtime!

An SQL Interface To std::vector

An SQL Interface To std::vector

It took less than a day to write a working (partial) SQL Interface for std::vector.

See https://github.com/rbock/sqlpp11-connector-stl

An SQL Interface To std::vector

```
#include "TabSample.h"
#include <sqlpp11/sqlpp11.h>
#include <sqlpp11/container/connection.h>

namespace sql = sqlpp::container;
constexpr TabSample tab{};

struct sample
{
  int64_t alpha;
  std::string beta;
  bool gamma;
};

using container = std::vector<sample>;
container data;
```

An SQL Interface To std::vector

```
sql::connection<container> db{data};

db(insert_into(tab).set(tab.alpha = 17));
db(insert_into(tab).set(tab.beta = "cheesecake"));
db(insert_into(tab).set(tab.alpha = 42,
                        tab.beta = "hello",
                        tab.gamma = true));

for (const sample& row: db(select(all_of(tab)).from(tab)
                                .where(tab.alpha < 18)))
{
  std::cerr << "alpha=" << row.alpha
            << ", beta=" << row.beta
            << ", gamma=" << row.gamma << std::endl;
}
```

What's next?

What's next?

sqlpp11 could be the foundation for type-safe SQL interfaces with immediate feedback at compile time for all kinds of databases, e.g.:

- SQL databases
- ODBC databases
- NoSQL databases
- Containers of the C++ Standard Library and others
- Streams
- XML
- Jason
- ...

What's next?

sqlpp11 could be the foundation for type-safe SQL interfaces with immediate feedback at compile time for all kinds of databases, e.g.:

- SQL databases
- ODBC databases
- NoSQL databases
- Containers of the C++ Standard Library and others
- Streams
- XML
- Jason
- ...

Your help is needed!

Acknowledgements

Thanks to everybody who uses the library, contributes to it, asks questions about it, listens to me when I talk about it, spreads the word. It really helps a lot to continuously improve the library.
In particular, I would like to thank (in chronological order of contact):

- Metafeed GmbH
- PPRO Financial Ltd
- The Boost community
- The ISO C++ Standards committee
- Meeting C++ Munich
- Battle For Wesnoth
- The audience