sqlpp11 - An SQL Library Worthy of Modern C++

Dr. Roland Bock

2014-09-11

Strings vs. EDSL

- *Prefer compile-time and link-time errors to runtime errors*
  Scott Meyers, Effective C++ (2nd Edition)

Strings vs. EDSL

Let's look at some code

String based

In the talk, we looked at a string-based example from cppdb first:
http://cppcms.com/sql/cppdb/intro.html
It is very easy to add all kinds of errors into this code that will pass the compiler but fail at runtime.

sqlpp11

We then looked at examples from sqlpp11:
https://github.com/rbock/sqlpp11/blob/develop/examples/insert.cpp
https://github.com/rbock/sqlpp11/blob/develop/examples/select.cpp
The compiler finds all those errors and more with sqlpp11 and reports them in a decent way. Check it out for yourself or watch the video.

The Member Template

Member Template

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

The Member Template

Member Template

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

Basic Usage

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

The Member Template

A real-world column

```
struct Feature
{
  struct _name_t
  {
    static constexpr const char* _get_name() { return "feature"; }
    template<typename T>
    struct _member_t
    {
      T feature;
      T& operator()() { return feature; }
      const T& operator()() const { return feature; }
    };
  };
  using _traits = sqlpp::make_traits<sqlpp::integer, sqlpp::tag::require_insert>;
};
```

The Member Template

A real-world table

```
struct TabPerson: sqlpp::table_t<TabPerson,
               TabPerson_::Id,
               TabPerson_::Name,
               TabPerson_::Feature>
{
  struct _name_t
  {
    static constexpr const char* _get_name() { return "tab_person"; }
    template<typename T>
    struct _member_t
    {
      T tabPerson;
      T& operator()() { return tabPerson; }
      const T& operator()() const { return tabPerson; }
    };
  };
};
```

The Member Template

Usage in tables

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

The Member Template

Usage in rows

```
template<typename Db, std::size_t index, typename FieldSpec>
struct result_field:
    public FieldSpec::_name_t::template
           _member_t<result_field_t<value_type_of<FieldSpec>, Db, FieldSpec>>
{
    //....
};
```

Constraints

I am *so* looking forward to Concepts Lite!

A real life example

Insert assignments

```
// Basics
static_assert(sizeof...(Assignments), "");
static_assert(all_t<is_assignment_t<Assignments>::value...>::value, "");
static_assert(not has_duplicates<lhs_t<Assignments>...>::value, "");

// All columns from one table
using _required_tables = make_joined_set_t<required_tables_of<lhs_t<Assignments>>..
static_assert(sizeof...(Assignments) ? (_required_tables::size::value == 1) : true,

// Table semantics required and prohibited insert columns
using _table = typename lhs_t<first_arg_t<Assignments...>>::_table;
using required_columns = typename _table::_required_insert_columns;
using columns = make_type_set_t<lhs_t<Assignments>...>;
static_assert(is_subset_of<required_columns, columns>::value, "");
static_assert(none_t<must_not_insert_t<lhs_t<Assignments>>::value...>::value, "");
```

Code Layers

Code using sqlpp11 has the following layers:

- user code
- sqlpp11 (vendor neutral)
- sqlpp11-connector
- native database library

Vendor Specific

Serialization

```
template<typename Select>
result_t select(const Select& x)
{
  _serializer_context_t context;
  ::sqlpp::serialize(x, context);
  return {...};
}
```

Vendor Specific

Serialization

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

Vendor Specific

Serialization

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

Vendor Specific

Disable a feature

```
template<typename Lhs, typename Rhs, typename On>
struct serializer_t<sqlite3::serializer_t, join_t<outer_join_t, Lhs, Rhs, On>>
{
    using T = join_t<outer_join_t, Lhs, Rhs, On>;

    static void _(const T& t, sqlite3::serializer_t& context)
    {
        static_assert(wrong_t<serializer_t>::value,
                      "Sqlite3: No support for outer join");
    }
};
```

Vendor Specific

Change the representation

```
template<typename First, typename... Args>
struct serializer_t<mysql::serializer_t, concat_t<First, Args...>>
{
    using T = concat_t<First, Args...>;

    static mysql::serializer_t& _(const T& t, mysql::serializer_t& context)
    {
        context << "CONCAT(";
        interpret_tuple(t._args, ',', context);
        context << ')';
        return context;
    }
};
```

Vendor Specific

What if I want something like this?

```
select(streets.name)
    .from(streets)
    .where(intersects(streets.geometry, some_polygon))

select(streets.name)
    .from(streets)
    .where(streets.geometry.within(from_wkt("POLYGON((0 0,10 0,10 10,0 10,0 0))")))

select(streets.name)
    .from(streets)
    .where(streets.geometry.distance(some_point) < 100)
```

(Examples by Adam Wulkiewicz)

Vendor Specific

Add a value type

```
struct integral
{
    using _traits = make_traits<integral, tag::is_value_type>;
    using _tag = tag::is_integral;
    using _cpp_value_type = int64_t;
};

template<typename Base>
struct expression_operators<Base, integral> { /*...*/ };

template<typename Base>
struct column_operators<Base, integral> { /*...*/ };

template<>
struct parameter_value_t<integral> { /*...*/ };

template<typename Db, typename FieldSpec>
struct result_field_t<integral, Db, FieldSpec> { /*...*/ };
```

Vendor Specific

Add a value method

```
template<typename T>
like_t<Base, wrap_operand_t<T>> like(T t) const
{
    using rhs = wrap_operand_t<T>;
    static_assert(_is_valid_operand<rhs>::value, "invalid argument for like()");

    return { *static_cast<const Base*>(this), {t} };
}
```

Vendor Specific

Add an expression node type

```
template<typename Operand, typename Pattern>
struct like_t:
    public expression_operators<like_t<Operand, Pattern>, boolean>,
    public alias_operators<like_t<Operand, Pattern>>
{
    using _traits = make_traits<boolean, tag::is_expression, tag::is_named_expression>;
    using _recursive_traits = make_recursive_traits<Operand, Pattern>;

    struct _name_t
    {
        static constexpr const char* _get_name() { return "LIKE"; }
        template<typename T>
            struct _member_t
            {
                T like;
                T& operator()() { return like; }
                const T& operator()() const { return like; }
            };
    };

    Operand _operand;
    Pattern _pattern;
};
```

Vendor Specific

Add a serializer

```
template<typename Context, typename Operand, typename Pattern>
struct serializer_t<Context, like_t<Operand, Pattern>>
{
    using T = like_t<Operand, Pattern>;

    static Context& _(const T& t, Context& context)
    {
        serialize(t._operand, context);
        context << " LIKE(";
        serialize(t._pattern, context);
        context << ")";
        return context;
    }
};
```

Vendor Specific

Add/Change/Remove subclauses

```
template<typename Database>
using blank_select_t = statement_t<Database,
            select_t,
            no_select_flag_list_t,
            no_select_column_list_t,
            no_from_t,
            no_where_t<true>,
            no_group_by_t,
            no_having_t,
            no_order_by_t,
            no_limit_t,
            no_offset_t>;

template<typename... Columns>
auto select(Columns... columns)
-> decltype(blank_select_t<void>().columns(columns...))
{
    return blank_select_t<void>().columns(columns...);
}
```

Use the interpreter

Unchartered territory

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

Use the interpreter

Unchartered territory

- Serialize into source code
  - A python script for extracting data from HTML
- Transform at compile-time into another expression tree

Use the interpreter

Reinterpreted Assignment

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

    Lhs _lhs;
    Rhs _rhs;
};

template<typename Lhs, typename Rhs>
struct interpreter_t<container::context_t, assignment_t<Lhs, Rhs>>
{
    static auto _(const assignment_t<Lhs, Rhs>& 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) };
        }
};
```

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

Code sample

```
constexpr TabSample tab{};

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

int main()
{
  sql::connection<std::vector<sample>> db{{}};

  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));
  db(insert_into(tab).set(tab.gamma = true));

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

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
- JSON
- ...

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.

Thank You!