Seventeenification

Porting sqlpp11 to C++17

Dr. Roland Bock

http://ppro.com
rbock at eudoxos dot de

https://github.com/rbock/sqlpp11
https://github.com/rbock/kiss-templates

MUC++, 2017-03-30

From C++98 to C++17

A little bit of history

2008 - String based queries

```
std::ostringstream os;
os << "SELECT * FROM tab_record "
   << "WHERE id > " << minId << " AND first_name == " << name
   << "ORDER BY priortiy DESC "
   << "LIMIT " << limit;
```

2008 - String based queries

```
std::ostringstream os;
os << "SELECT * FROM tab_record "
   << "WHERE id > " << minId << " AND first_name == " << name  // Missing space
   << "ORDER BY priortiy DESC "
   << "LIMIT " << limit;
```

2010 - RFC for SQL EDSL on boost mailing list

```
typedef my_table<> t;
typedef sql::select_record<t> record;

//...

std::vector<record> records =
     db.select<record>(
         sql::where(t::id() > 1000 && t::first_name() == name),
         sql::order_by(t::priority()(sql::desc)),
         sql::limit(17));
```

http://lists.boost.org/Archives/boost/2010/09/170947.php

2013 - RFC for sqlpp11 on boost mailing list

```
for (const auto& row : db.run(
                        select(foo.name, foo.hasFun)
                        .from(foo)
                        .where(foo.id > 17 and foo.name.like("%bar%"))))
{
  if (row.name.is_null())
    std::cerr << "name will convert to empty string" << std::endl;
  std::string name = row.name;

  bool hasFun = row.hasFun;

}
```

http://lists.boost.org/Archives/boost/2013/11/208388.php

2014 - 2016 Several talks at MUC++, MeetingC++ and CppCon

```
sqlpp11/where.h:212:3: error: static_assert failed "calling where() or unconditionally() required"
  SQLPP_PORTABLE_STATIC_ASSERT(assert_where_or_unconditionally_called_t,
  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

/home/rbock/projects/sqlpp11/include/sqlpp11/portable_static_assert.h:41:7: note: expanded from macro 'S
      static_assert(wrong_t<T...>::value, message); \
      ^             ~~~~~~~~~~~~~~~~~~~~

tests/MockDb.h:125:20: note: in instantiation of function template specialization 'sqlpp::assert_where_o
    return _run(t, sqlpp::run_check_t<_serializer_context_t, T>{});
                   ^

tests/Select.cpp:83:19: note: in instantiation of function template specialization 'MockDbT<false>::oper
    auto rows = db(select(all_of(t)).from(t));
```

2016 - Variants of variadic AND at CppCon

2016 - Variants of variadic AND at CppCon

```
template<bool...>
struct all_helper
{};

template<bool... Args>
using all_11 = std::is_same<all_helper<true, Args...>,
                            all_helper<Args..., true>>;
```

2016 - Variants of variadic AND at CppCon

```
template<bool...>
struct all_helper
{};

template<bool... Args>
using all_11 = std::is_same<all_helper<true, Args...>,
                            all_helper<Args..., true>>;

template<bool... Args>
constexpr auto all_17 = (true && ... && Args);
```

https://www.youtube.com/watch?v=VNrShPCgVjw

From C++98 to C++17

Let's write sqlpp17

Inline variables and auto non-type template parameters

Inline variables and auto non-type template parameters

A column definition for sqlpp11

```
struct FirstName
{
  struct _alias_t
  {
    static constexpr const char _literal[] = "first_name";

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

Inline variables and auto non-type template parameters

This does not link with C++11

```
#include <iostream>

struct foo
{
  static constexpr const char bar[] = "bar";
};

int main()
{
  std::cout << foo::bar;
}
```

Inline variables and auto non-type template parameters

char_sequence to the rescue

```
template <char... Cs>
struct char_sequence
{
  static const char* char_ptr()
  {
    static char s[] = {Cs...};
    return s;
  };
};
```

Inline variables and auto non-type template parameters

Constructing a char_sequence

```
template <std::size_t N, const char (&Input)[N]>
using make_char_sequence = typename make_char_sequence_impl<
    N, Input, sqlpp::detail::make_index_sequence<N>>::type;
```

Inline variables and auto non-type template parameters

Constructing a char_sequence

```
template <std::size_t N, const char (&s)[N], typename T>
struct make_char_sequence_impl;
```

```
template <std::size_t N, const char (&Input)[N]>
using make_char_sequence = typename make_char_sequence_impl<
    N, Input, sqlpp::detail::make_index_sequence<N>>::type;
```

Inline variables and auto non-type template parameters

Constructing a char_sequence

```
template <std::size_t N, const char (&s)[N], typename T>
struct make_char_sequence_impl;

template <std::size_t N, const char (&s)[N], std::size_t... i>
struct make_char_sequence_impl<N, s, sqlpp::detail::index_sequence<i...>>
{
  using type = char_sequence<s[i]...>;
};

template <std::size_t N, const char (&Input)[N]>
using make_char_sequence = typename make_char_sequence_impl<
    N, Input, sqlpp::detail::make_index_sequence<N>>::type;
```

Inline variables and auto non-type template parameters

A column definition for sqlpp11

```
struct FirstName
{
  struct _alias_t
  {
    static constexpr const char _literal[] = "first_name";
    using _name_t = sqlpp::make_char_sequence<sizeof(_literal), _literal>;
  };
};
```

Inline variables and auto non-type template parameters

This compiles and links with C++17!

```
#include <iostream>

struct foo
{
  static constexpr const char bar[] = "bar";
};

int main()
{
  std::cout << foo::bar;
}
```

Inline variables and auto non-type template parameters

Constructing a char_sequence

```
template <const auto& StringLiteral>
using make_char_sequence = typename make_char_sequence_impl<StringLiteral>::type;
```

Inline variables and auto non-type template parameters

Constructing a char_sequence

```
template <const auto& Value>
struct make_char_sequence_impl;



template <const auto& StringLiteral>
using make_char_sequence = typename make_char_sequence_impl<StringLiteral>::type;
```

Inline variables and auto non-type template parameters

Constructing a char_sequence

```
template <const auto& Value>
struct make_char_sequence_impl;

template <std::size_t N, const char (&StringLiteral)[N]>
struct make_char_sequence_impl<StringLiteral>;

template <const auto& StringLiteral>
using make_char_sequence = typename make_char_sequence_impl<StringLiteral>::type;
```

Inline variables and auto non-type template parameters

Comparison

```
struct _alias_t
{
  static constexpr const char _literal[] = "first_name";
  using _name_t = sqlpp::make_char_sequence<sizeof(_literal), _literal>;
};
```

versus

```
struct _alias_t
{
  static constexpr const char _literal[] = "first_name";
  using _name_t = sqlpp::make_char_sequence<_literal>;
};
```

Inline variables and auto non-type template parameters

Construct the char_sequence externally

```
template <typename Table, typename ColumnSpec>
struct char_sequence_of<column_t<Table, ColumnSpec>>
{
  using type = make_char_sequence<ColumnSpec::_alias_t::_literal>;
};
```

Inline variables and auto non-type template parameters

A column definition for sqlpp17 [update]

```
struct FirstName
{
  struct _alias_t
  {
    static constexpr const char _name[] = "first_name";
  };
};
```

Inline variables and auto non-type template parameters

Migration result

- Nicer code (construction of name types)
- Less code (usable inline literals)
- Improved compile time (less name types)
- Maybe some backports to sqlpp11

[[nodiscard]], if constexpr, and class template deduction

[[nodiscard]], if constexpr, and class template deduction

Bad usage of sqlpp11

```
auto s = select(all_of(t))
         .from(t)
         .where(t.id > 17);
```

[[nodiscard]], if constexpr, and class template deduction

Bad usage of sqlpp11

```
auto s = select(all_of(t))
         .from(t)
         .where(t.id > 17);

s.order_by(t.name.asc());
```

[[nodiscard]], if constexpr, and class template deduction

The order_by function in sqlpp11

```
template <typename... Expressions>
auto order_by(Expressions... expressions) const
    -> _new_statement_t<check_order_by_t<Expressions...>, order_by_t<void, Expressions...>>
{
  return _order_by_impl(check_order_by_t<Expressions...>{}, expressions...);
}
```

[[nodiscard]], if constexpr, and class template deduction

The order_by function in sqlpp11

```
template <typename... Expressions>
auto order_by(Expressions... expressions) const
    -> _new_statement_t<check_order_by_t<Expressions...>, order_by_t<void, Expressions...>>
{
  return _order_by_impl(check_order_by_t<Expressions...>{}, expressions...);
}

template <typename... Expressions>
auto _order_by_impl(consistent_t, Expressions... expressions) const
    -> _new_statement_t<consistent_t, order_by_t<Expressions...>>;
```

[[nodiscard]], if constexpr, and class template deduction

The order_by function in sqlpp11

```
template <typename... Expressions>
auto order_by(Expressions... expressions) const
    -> _new_statement_t<check_order_by_t<Expressions...>, order_by_t<void, Expressions...>>
{
  return _order_by_impl(check_order_by_t<Expressions...>{}, expressions...);
}

template <typename... Expressions>
auto _order_by_impl(consistent_t, Expressions... expressions) const
    -> _new_statement_t<consistent_t, order_by_t<Expressions...>>;

template <typename Check, typename... Expressions>
auto _order_by_impl(Check, Expressions... expressions) const
    -> inconsistent<Check>;
```

[[nodiscard]], if constexpr, and class template deduction

The order_by function in sqlpp17

```
template <typename... Expressions>
[[nodiscard]] constexpr auto order_by(Expressions... expressions) const
{
  constexpr auto check = check_order_by_arg(expressions...);

  if constexpr (check)
  {
    return Statement::of(this).replace_clause(no_order_by_t{},
        order_by_t<Expressions...>{std::make_tuple(expressions...)});
  }
  else
  {
    return ::sqlpp::bad_statement_t<std::decay_t<decltype(check)>>{};
  }
}
```

[[nodiscard]], if constexpr, and class template deduction

The order_by function in sqlpp17

```
template <typename... Expressions>
[[nodiscard]] constexpr auto order_by(Expressions... expressions) const
{
  constexpr auto check = check_order_by_arg(expressions...);

  if constexpr (check)
  {
    return Statement::of(this).replace_clause(no_order_by_t{},
        order_by_t{std::make_tuple(expressions...)});
  }
  else
  {
    return ::sqlpp::bad_statement_t{check};
  }
}
```

[[nodiscard]], if constexpr, and class template deduction

The bad statement

```
template <typename Failure>
struct bad_statement_t
{
  constexpr bad_statement_t()
  {
    static_assert(wrong<Failure>, "Missing specialization");
  }

  constexpr bad_statement_t(const Failure);
};
```

[[nodiscard]], if constexpr, and class template deduction

Bad usage of sqlpp17

```
auto s = select(all_of(t))
         .from(t)
         .where(t.id > 17);

s.order_by(t.name.asc());
```

This is now a compile time warning (almost certainly).

[[nodiscard]], if constexpr, and class template deduction

Migration result

- Much nicer code (no tag dispatch)
- Less code (no tag dispatch, no additional function declarations)
- Cleaner code (class template deduction)
- Improved compile time (less templates)?
- Less bugs (nodiscard)
- No backports to sqlpp11

fold expressions

fold expressions

Variadic all using C++17

```
template<bool... Args>
constexpr auto all = (true && ... && Args);
```

fold expressions

Print tuple members as comma separated list.

A small helper

```
struct separator
{
  std::ostream& os;
  const char* sep;
  bool is_first = true;

  template <typename Expr>
  decltype(auto) operator()(const Expr& expr)
  {
    if (is_first)
      is_first = false;
    else
      os << sep;
    return expr;
  }
};
```

fold expressions

Printing tuples with C++17

```
template <typename... Columns>
decltype(auto) operator<<(std:ostream& os, const std::tuple<Columns...>& t)
{
  auto separate = detail::separator{os, ", "};
  return (os << ... << separate(std::get<Columns>(t)));
}
```

fold expressions

A typical compile time check in sqlpp:
Are the column names in a select unique?

Type set basics

```
template <typename T>
struct _base {};
```

Type set basics

```
template <typename T>
struct _base {};

template <typename... Elements>
struct _type_set
{
private:
  struct _impl : _base<Elements>...
  {
  };
```

Type set basics

```
template <typename T>
struct _base {};

template <typename... Elements>
struct _type_set
{
private:
  struct _impl : _base<Elements>...
  {
  };

public:
  template <typename T>
  [[nodiscard]] static constexpr auto count()
  {
    return std::is_base_of<_base<T>, _impl>::value;
  }
```

Type set comparison

```
template <typename... Elements>
struct _type_set
{
  template <typename... T>
  [[nodiscard]] constexpr auto operator>=(_type_set<T...>) const
  {
    return (true && ... && count<T>());
  }
```

Insert into a type set

```
template <typename... Elements>
struct _type_set
{
  template <typename T>
  [[nodiscard]] static constexpr auto insert()
  {
    return std::conditional_t<count<T>(), _type_set, _type_set<Elements..., T>>{};
  }
```

Insert into a type set

```
template <typename... Elements>
struct _type_set
{
  template <typename T>
  [[nodiscard]] static constexpr auto insert()
  {
    return std::conditional_t<count<T>(), _type_set, _type_set<Elements..., T>>{};
  }

  template <typename T>
  [[nodiscard]] constexpr auto operator<<(_base<T>) const
  {
    return insert<T>();
  }
```

fold expressions

Constructing a type set

```
template <typename... Ts>
constexpr auto type_set()
{
  return (detail::_type_set{} << ... << detail::_base<Ts>{});
}
```

Testing for uniqueness

```
template <typename... T>
constexpr auto check_select_columns_arg(const T&...)
{
  if constexpr(type_set<char_sequence_of_t<T>...>().size() != sizeof...(T))
  {
    return failed<assert_select_columns_args_have_unique_names>{};
  }
  else
    return succeeded{};
}
```

Migration result

- More expressive code
- Less code
- Improved compile time (less recursion)?
- Improved compile time (reduced golden hammer syndrom)!
- Some backports to sqlpp11

std::string_view

Text result field in sqlpp11

```
template <...>
class text_result_field_t<...>
{
  const char* text{nullptr};  // Non-owning
  size_t len{};

public:
  std::string value;

  template <typename Target>
  void _bind(Target& target, size_t index)
  {
    target._bind_result(index, &text, &len);
    value.assign(text, len);
  }
};
```

std::string_view

Text result field in sqlpp17

```
template <...>
class text_result_field_t<...>
{
  const char* text{nullptr};  // Non-owning
  size_t len{};

public:
  std::string_view value;

  template <typename Target>
  void _bind(Target& target, size_t index)
  {
    target._bind_result(index, &text, &len);
    value = std::string_view{text, len};
  }
};
```

std::string_view

Migration result

- Improved runtime performance
- No backport to sqlpp11

std::variant and std::optional

std::optional

std::variant and std::optional

A long-standing demand

If result field or a query parameter can be NULL, model them as std::optional.

std::variant and std::optional

Writing a query with std::optional can be annoying

```
auto s = select(all_of(t)).from(t).where(
  t.id > 17 and
  t.foo == (value == 0
            ? std::optional<int64_t>{}
            : std::optional<int64_t>{value}));
```

std::variant and std::optional

Thus, in sqlpp17, the use of std::optional is optional.

The classic demand for std::optional

Writing a query with std::optional

```
template <typename FieldSpec, bool CanBeNull, bool NullIsTrivialValue>
struct make_result_field_base;
```

The classic demand for std::optional

Writing a query with std::optional

```
template <typename FieldSpec, bool CanBeNull, bool NullIsTrivialValue>
struct make_result_field_base;

template <typename FieldSpec>
struct make_result_field_base<FieldSpec, true, true>
{
  using type = member_t<FieldSpec, null_is_trivial<cpp_type_of_t<value_type_of_t<FieldSpec>>>>;
};
```

The classic demand for std::optional

Writing a query with std::optional

```
template <typename FieldSpec, bool CanBeNull, bool NullIsTrivialValue>
struct make_result_field_base;

template <typename FieldSpec>
struct make_result_field_base<FieldSpec, true, true>
{
  using type = member_t<FieldSpec, null_is_trivial<cpp_type_of_t<value_type_of_t<FieldSpec>>>>;
};

template <typename FieldSpec>
struct make_result_field_base<FieldSpec, true, false>
{
  using type = member_t<FieldSpec, std::optional<cpp_type_of_t<value_type_of_t<FieldSpec>>>>;
};
```

std::variant and std::optional

std::variant

Dynamic queries in sqlpp11

```
auto s = dynamic_select(db).dynamic_columns(foo.id).dynamic_from(foo).unconditionally();
```

Dynamic queries in sqlpp11

```
auto s = dynamic_select(db).dynamic_columns(foo.id).dynamic_from(foo).unconditionally();

if (someCondition)
  s.selected_columns.add(foo.name);
```

Dynamic queries in sqlpp11

```
auto s = dynamic_select(db).dynamic_columns(foo.id).dynamic_from(foo).unconditionally();

if (someCondition)
  s.selected_columns.add(foo.name);

if (someOtherCondition)
{
  s.selected_columns.add(bar.hasFun);
  s.from.add(dynamic_join(bar).on(foo.barId == bar.id));
}
```

Accessing results of dynamic queries in sqlpp11

```
for (const auto& row : s)
{
  std::cout << row.id;
  if (someCondition)
      std::cout << row["name"];
  if (someOtherCondition)
      std::cout << row["hasFun"];
}
```

std::variant and std::optional

New sqlpp17 example

```
for (const auto& row : select(foo.id,
                              make_optional(someCondition, foo.name),
                              make_optional(someOtherCondition, bar.hasFun))
                       .from(foo
                             .join(make_optional(someOtherCondition, bar))
                             .on(foo.barId == bar.id))
                       .unconditionally())
{
  std::cout << row.id;
  if (someCondition) std::cout << row.name;
  if (someOtherCondition) std::cout << row.hasFun;
}
```

std::variant and std::optional

Using a tuple of std::optional we

- know all potential tables and columns and their names at compile time
- can obtain all result fields as data members of a struct

std::variant and std::optional

However, it turns out to be easier to use something like this inside the library:

sqlpp::optional

```
template <typename T>
struct optional
{
  bool to_be_used;
  T value;
};
```

However, it turns out to be easier to use something like this inside the library:

sqlpp::optional

```
template <typename T>
struct optional
{
  bool to_be_used;
  T value;
};
```

And this can also be used with C++11.

std::variant and std::optional

Migration result

- Optional use of std::optional for fields and parameters.
- Thinking about them leads to nice new ideas.
- Some backports possible.

Summary

From my personal sqlpp17-perspective

auto non-type template parameters	quite nice
class template deduction	sweet syntactic sugar
inline variables	awesome
[[nodiscard]]	stellar
if constexpr	cooler than ice cream
fold expressions	fantabulous
string_view	wonderful
optional	helpful
variant	not for me (yet)

Summary

From my personal sqlpp17-perspective

auto non-type template parameters	quite nice
class template deduction	sweet syntactic sugar
inline variables	awesome
[[nodiscard]]	stellar
if constexpr	cooler than ice cream
fold expressions	fantabulous
string_view	wonderful
optional	helpful
variant	not for me (yet)

Experimenting with C++17 helps to improve my C++11 code.

Questions?

Thank you!