Pruning Error Messages From Your C++ Template Code

Roland Bock

rbock eudoxos de

https://github.com/rbock/sqlpp11

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

CppCon, 2015-09-24

What got me started

Expressive C++:
Why Template Errors Suck and What You Can Do About It

Eric Niebler, C++Next, 2010

Setting the stage

Code samples are taken from sqlpp11

```
for (const auto& row : db(select(foo.name, foo.hasFun)
                          .from(foo)
                          .where(foo.id > 17 and foo.name.like("%bar%"))))
{
    std::cerr << row.name << ", " << row.hasFun << '\n';
}
```

Compilers

A simple mistake

```
for (const auto& row : db(select(t.alpha).from("t").where(true)))
{
    std::cerr << row.alpha << std::endl;
}
```

On the 0.25 release of sqlpp11 this example produces

- 18kb of error messages in 100 lines for gcc-5.0
- 38kb of error messages in 292 lines for clang-3.6

No silver bullet

In this talk I will present techniques to reduce the amount of template error messages. They will range from

- *"Easy to explain, but hard to do"* to
- *"Hard to explain, and hard to do"*

Nothing will help in all situations.

Compilers

A simple mistake

```
for (const auto& row : db(select(t.alpha).from("t").where(true)))
{
    std::cerr << row.alpha << std::endl;
}
```

On the 0.25 release of sqlpp11 this example produces

- 18kb of error messages in 100 lines for gcc-5.0
- 38kb of error messages in 292 lines for clang-3.6

Compilers

Compilers make a difference. . .

. . . so you should support compiler developers.

Libraries

The quality of code makes a difference. . .

. . . so you should support library developers.

The best way to avoid template errors. . .

Don't use templates.

Don't use templates.
No kidding.

Don’t use templates

- Make sure that you *really* need to use templates
- Make sure that you *really* need to expose templates in your API
- Maybe runtime polymorphism is completely sufficient
- Maybe you can hide your templates

Don't use templates

Hiding a template

```
struct TabBar: public sqlpp::table_t<TabBar,
                                     TabBar_::Alpha,
                                     TabBar_::Beta,
                                     TabBar_::Gamma,
                                     TabBar_::Delta>
{
  //...
};
```

Don't use templates

Hiding a template

```
struct TabBar: public sqlpp::table_t<TabBar,
                                     TabBar_::Alpha,
                                     TabBar_::Beta,
                                     TabBar_::Gamma,
                                     TabBar_::Delta>
{
  //...
};
```

Inheritance effectively hides the template.

Don't use templates

All templates visible

```
tests/SelectTest.cpp:44:4: note: in instantiation of
      function template specialization
      'MockDbT<false>::operator()<sqlpp::statement_t<void, sqlpp::select_t,
      sqlpp::no_select_flag_list_t, sqlpp::select_column_list_t<void,
      sqlpp::column_t<sqlpp::table_t<test::_TabBar, test::TabBar_::Alpha_,
      test::TabBar_::Beta_, test::TabBar_::Gamma_, test::TabBar_::Delta_>,
      test::TabBar_::Alpha_>, sqlpp::column_t<sqlpp::table_t<test::_TabBar,
      test::TabBar_::Alpha_, test::TabBar_::Beta_, test::TabBar_::Gamma_,
      test::TabBar_::Delta_>, test::TabBar_::Beta_>,
      sqlpp::column_t<sqlpp::table_t<test::_TabBar, test::TabBar_::Alpha_,
      test::TabBar_::Beta_, test::TabBar_::Gamma_, test::TabBar_::Delta_>,
      test::TabBar_::Gamma_>, sqlpp::column_t<sqlpp::table_t<test::_TabBar,
      test::TabBar_::Alpha_, test::TabBar_::Beta_, test::TabBar_::Gamma_,
      test::TabBar_::Delta_>, test::TabBar_::Delta_> >, sqlpp::from_t<void,
      sqlpp::table_t<test::_TabBar, test::TabBar_::Alpha_, test::TabBar_::Beta_,
      test::TabBar_::Gamma_, test::TabBar_::Delta_> >, sqlpp::no_extra_tables_t,
      sqlpp::no_where_t<true>, sqlpp::no_group_by_t, sqlpp::no_having_t,
      sqlpp::no_order_by_t, sqlpp::no_limit_t, sqlpp::no_offset_t> >' requested
      here
        db(select(all_of(t)).from(t));
```

Don’t use templates

After hiding table templates behind inheritance

```
tests/SelectTest.cpp:44:4: note: in instantiation
      of function template specialization
      'MockDbT<false>::operator()<sqlpp::statement_t<void, sqlpp::select_t,
      sqlpp::no_select_flag_list_t, sqlpp::select_column_list_t<void,
      sqlpp::column_t<test::TabBar, test::TabBar_::Alpha>,
      sqlpp::column_t<test::TabBar, test::TabBar_::Beta>,
      sqlpp::column_t<test::TabBar, test::TabBar_::Gamma>,
      sqlpp::column_t<test::TabBar, test::TabBar_::Delta> >,
      sqlpp::from_t<void, test::TabBar>, sqlpp::no_extra_tables_t,
      sqlpp::no_where_t<true>, sqlpp::no_group_by_t, sqlpp::no_having_t,
      sqlpp::no_order_by_t, sqlpp::no_limit_t, sqlpp::no_offset_t> >'
      requested here
        db(select(all_of(t)).from(t));
```

Don't use templates

After hiding column templates behind inheritance

```
tests/SelectTest.cpp:44:4: note: in instantiation
      of function template specialization
      'MockDbT<false>::operator()<sqlpp::statement_t<void, sqlpp::select_t,
      sqlpp::no_select_flag_list_t, sqlpp::select_column_list_t<void,
      test::TabBar_::Alpha_, test::TabBar_::Beta_, test::TabBar_::Gamma_,
      test::TabBar_::Delta_ >, sqlpp::from_t<void, test::TabBar>,
      sqlpp::no_extra_tables_t, sqlpp::no_where_t<true>, sqlpp::no_group_by_t,
      sqlpp::no_having_t, sqlpp::no_order_by_t, sqlpp::no_limit_t, sqlpp::no_offse
      requested here
        db(select(all_of(t)).from(t));
```

Don't use templates

Don't use templates if you don't have to.

Try to avoid recursion

Try to avoid recursion

Easier said than done.

Try to avoid recursion

Easier said than done.

But

- Recursion is slow.
- Recursion is limited.
- Recursion leads to dreadful error messages.

Try to avoid recursion

Call f for each argument

```
template<typename F, typename... Args>
void call_for_each(F&& f, Args&&... args);
```

Try to avoid recursion

A recursive version

```
template<typename F>
void call_for_each(F&& f)
{
}
```

Try to avoid recursion

A recursive version

```
template<typename F>
void call_for_each(F&& f)
{
}

template<typename F, typename Arg, typename... Rest>
void call_for_each(F&& f, Arg&& arg, Rest&&... rest)
{
      f(std::forward<Arg>(arg));
      call_for_each(std::forward<F>(f), std::forward<Rest>(rest)...);
}
```

Try to avoid recursion

Let's call it

```
void test(int i)
{
  std::cerr << i;
}

int main()
{
  const auto ten = std::map<std::string, std::vector<int>>{};
  call_for_each(test, 1, 2, 3, 4, 5, 6, 7, 8, 9, ten);
}
```

Try to avoid recursion

Compile with clang-3.6

```
for_each_arg.cpp:23:4: error: no viable conversion from 'const std::map<std::basic_string<char>, std::vector<i
      std::basic_string<char>, std::vector<int, std::allocator<int> > > > >' to 'int'
        f(arg);
          ^~~
for_each_arg.cpp:24:2: note: in instantiation of function template specialization 'call_for_each<void (&)(int)
      std::less<std::basic_string<char> >, std::allocator<std::pair<const std::basic_string<char>, std::vector
        call_for_each(f, rest...);
        ^
for_each_arg.cpp:24:2: note: in instantiation of function template specialization 'call_for_each<void (&)(int)
      std::less<std::basic_string<char> >, std::allocator<std::pair<const std::basic_string<char>, std::vector
        call_for_each(f, rest...);
        ^
for_each_arg.cpp:24:2: note: in instantiation of function template specialization 'call_for_each<void (&)(int)
      std::less<std::basic_string<char> >, std::allocator<std::pair<const std::basic_string<char>, std::vector
        call_for_each(f, rest...);
        ^
for_each_arg.cpp:24:2: note: in instantiation of function template specialization 'call_for_each<void (&)(int)
      std::less<std::basic_string<char> >, std::allocator<std::pair<const std::basic_string<char>, std::vector
        call_for_each(f, rest...);
        ^
for_each_arg.cpp:24:2: note: in instantiation of function template specialization 'call_for_each<void (&)(int)
      std::allocator<int> >, std::less<std::basic_string<char> >, std::allocator<std::pair<const std::basic_st
        call_for_each(f, rest...);
        ^
for_each_arg.cpp:24:2: note: in instantiation of function template specialization 'call_for_each<void (&)(int)
      std::allocator<int> >, std::less<std::basic_string<char> >, std::allocator<std::pair<const std::basic_st
        call_for_each(f, rest...);
        ^
for_each_arg.cpp:24:2: note: in instantiation of function template specialization 'call_for_each<void (&)(int)
      std::allocator<int> >, std::less<std::basic_string<char> >, std::allocator<std::pair<const std::basic_st
```

Try to avoid recursion

A non-recursive version

```
template<typename F, typename... Args>
void call_for_each(F&& f, Args&&... args)
{
  using swallow = int[];
  (void)swallow{(f(std::forward<Args>(args)), 0)...};
}
```

Try to avoid recursion

Compile with clang-3.6

```
for_each_arg.cpp:12:19: error: no viable conversion from 'const std::map<std::basi
      std::basic_string<char>, std::vector<int, std::allocator<int> > > > >' to 'i
        (void)swallow{(f(std::forward<Args>(args)), 0)...};

for_each_arg.cpp:37:2: note: in instantiation of function template specialization
      std::vector<int, std::allocator<int> >, std::less<std::basic_string<char> >,
        call_for_each(test, 1, 2, 3, 4, 5, 6, 7, 8, 9, ten);
1 error generated.
```

Try to avoid recursion

C++1z style (fold expressions, N4295)

```
template<typename F, typename... Args>
void call_for_each(F&& f, Args&&... args)
{
  (f(std::forward<Args>(args)),...);
}
```

Try to avoid recursion

A non-recursive all

Try to avoid recursion

A non-recursive all

```
template<bool... B>
struct logic_helper;
```

Try to avoid recursion

A non-recursive all

```
template<bool... B>
struct logic_helper;

template<bool... B>
using all_t = std::integral_constant<
        bool,
        std::is_same<logic_helper<B...>, logic_helper<(B, true)...>>::value>;
```

Try to avoid recursion

A non-recursive all, C++17 style

```
template<bool... B>
using all_t = std::integral_constant<bool, true && ... && B>;
```

Using static_assert

Using static_assert

Basic idea

```
template<typename T>
struct my_integral_handler
{
    static_assert(std::is_integral<T>::value,
                  "Integral template parameter required");

    // ...
};
```

Using static_assert

Partial specializations

```
template<typename L, typename R>
struct joined_type_set;
```

Using static_assert

Partial specializations

```
template<typename L, typename R>
struct joined_type_set;

template<typename... Ls, typename... Rs>
struct joined_type_set<type_set<Ls...>, type_set<Rs...>>
{
    using type = typename make_type_set<Ls..., Rs...>::type;
};
```

Using static_assert

Partial specializations

```
template<typename L, typename R>
struct joined_type_set;

template<typename... Ls, typename... Rs>
struct joined_type_set<type_set<Ls...>, type_set<Rs...>>
{
    using type = typename make_type_set<Ls..., Rs...>::type;
};

template<typename L, typename R>
struct joined_type_set;
{
    static_assert(???, "L and R have to be type sets");
};
```

Using static_assert

Partial specializations

```
template<typename L, typename R>
struct joined_type_set;

template<typename... Ls, typename... Rs>
struct joined_type_set<type_set<Ls...>, type_set<Rs...>>
{
    using type = typename make_type_set<Ls..., Rs...>::type;
};

template<typename L, typename R>
struct joined_type_set;
{
    static_assert(???, "L and R have to be type sets");
};
```

A "just-in-time false" would be great...

Using static_assert

A "just-in-time false"

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

Using static_assert

A "just-in-time false"

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

template<typename L, typename R>
struct joined_type_set
{
    static_assert(wrong_t<joined_type_set>::value, "L and R have to be type sets");
};
```

Using static_assert

A "just-in-time false"

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

template<typename L, typename R>
struct joined_type_set
{
    static_assert(wrong_t<joined_type_set>::value, "L and R have to be type sets");
};

template<typename... Ls, typename... Rs>
struct joined_type_set<type_set<Ls...>, type_set<Rs...>>
{
    using type = typename make_type_set<Ls..., Rs...>::type;
};
```

Using static_assert

Example of static assert in a function

```
template<typename... Tables>
auto from(Tables... tables) const
-> _new_statement_t<from_t<void, Tables...>>
{
  return _from_impl<void>(tables...);
}
```

Using static_assert

Example of static assert in a function

```
template<typename... Tables>
auto from(Tables... tables) const
-> _new_statement_t<from_t<void, Tables...>>
{
  return _from_impl<void>(tables...);
}

template<typename Database, typename... Tables>
auto _from_impl(Tables... tables) const
-> _new_statement_t<from_t<Database, Tables...>>
{
  static_assert(all_t<is_table_t<Tables>::value...>::value,
        "at least one argument is not a table or join in from()");
  //...
}
```

Using static_assert

Error output for incorrect from() call

```
include/sqlpp11/from.h:189:7: error: static_assert failed "at least one argument is not a table or join in fro
                            static_assert(all_t<is_table_t<Tables>::value...>::value, "at least one argument i

include/sqlpp11/from.h:173:14: note: in instantiation of function template specialization
      'sqlpp::no_from_t::_methods_t<sqlpp::detail::statement_policies_t<void, sqlpp::select_t, sqlpp::no_selec
      sqlpp::select_column_list_t<void, test::TabBar_::Alpha, test::TabBar_::Beta, test::TabBar_::Gamma,
      test::TabBar_::Delta>, sqlpp::no_from_t, sqlpp::no_extra_tables_t, sqlpp::no_where_t<true>, sqlpp::no_gr
      sqlpp::no_having_t, sqlpp::no_order_by_t, sqlpp::no_limit_t, sqlpp::no_offset_t> >::_from_impl<void,
      sqlpp::column_t<test::TabBar, test::TabBar_::Alpha> >' requested here
                                                return _from_impl<void>(tables...);

tests/SelectTest.cpp:44:20: note: in instantiation of function template specialization
      'sqlpp::no_from_t::_methods_t<sqlpp::detail::statement_policies_t<void, sqlpp::select_t, sqlpp::no_selec
      test::TabBar_::Alpha, test::TabBar_::Beta, test::TabBar_::Gamma, test::TabBar_::Delta>, sqlpp::no_from_t
      sqlpp::no_group_by_t, sqlpp::no_having_t, sqlpp::no_order_by_t, sqlpp::no_limit_t,
      sqlpp::no_offset_t> >::from<sqlpp::column_t<test::TabBar, test::TabBar_::Alpha> >' requested here
        select(all_of(t)).from(t.alpha);
```

Using static_assert

Error output for incorrect from() call

```
include/sqlpp11/from.h:189:7: error: static_assert failed "at least one argument is not a table or join in fro
                               static_assert(all_t<is_table_t<Tables>::value...>::value, "at least one argument i

include/sqlpp11/from.h:173:14: note: in instantiation of function template specialization
      'sqlpp::no_from_t::_methods_t<sqlpp::detail::statement_policies_t<void, sqlpp::select_t, sqlpp::no_selec
      sqlpp::select_column_list_t<void, test::TabBar_::Alpha, test::TabBar_::Beta, test::TabBar_::Gamma,
      test::TabBar_::Delta>, sqlpp::no_from_t, sqlpp::no_extra_tables_t, sqlpp::no_where_t<true>, sqlpp::no_gr
      sqlpp::no_having_t, sqlpp::no_order_by_t, sqlpp::no_limit_t, sqlpp::no_offset_t> >::_from_impl<void,
      sqlpp::column_t<test::TabBar, test::TabBar_::Alpha> >' requested here
                                                     return _from_impl<void>(tables...);

tests/SelectTest.cpp:44:20: note: in instantiation of function template specialization
      'sqlpp::no_from_t::_methods_t<sqlpp::detail::statement_policies_t<void, sqlpp::select_t, sqlpp::no_selec
      test::TabBar_::Alpha, test::TabBar_::Beta, test::TabBar_::Gamma, test::TabBar_::Delta>, sqlpp::no_from_t
      sqlpp::no_group_by_t, sqlpp::no_having_t, sqlpp::no_order_by_t, sqlpp::no_limit_t,
      sqlpp::no_offset_t> >::from<sqlpp::column_t<test::TabBar, test::TabBar_::Alpha> >' requested here
        select(all_of(t)).from(t.alpha);


In file included from tests/SelectTest.cpp:31:
In file included from include/sqlpp11/select.h:36:
include/sqlpp11/from.h:190:7: error: static_assert failed "at least one table depends on another table"
                               static_assert(required_tables_of<from_t<Database, Tables...>>::size::value == 0, "
```

Using static_assert

That example did not work out too well

- The static assert is too far away from the call site.
- The compiler continues to splatter after being hit by a static assert.
- With a static assert or other hard errors in the return type, you get many times more junk.
- With a non-sqlpp11 argument in from(), you get more than a hundred lines of unsolicitated spew.

Let's try enable_if and Concepts Lite

Using enable_if

Let's try enable_if and Concepts Lite

Using enable_if

```
template<typename... Tables>
auto from(Tables... tables) const
-> typename std::enable_if<
      all_t<is_table_t<Tables>::value...>::value,
      _new_statement_t<from_t<void, Tables...>>
      >::type
{
    //...
}
```

Let's try enable_if and Concepts Lite

clang's error message for enable_if

```
tests/SelectTest.cpp:44:18: error: no matching member
      function for call to 'from'
        select(t.alpha).from(t.alpha);

include/sqlpp11/from.h:180:6: note: candidate template
      ignored: disabled by 'enable_if' [with Tables = <test::TabBar_::Alpha_>>]
                    all_t<is_table_t<Tables>::v...
```

Beautiful.

Let's try enable_if and Concepts Lite

gcc's error message for enable_if

```
tests/SelectTest.cpp: In function  int main()  :
tests/SelectTest.cpp:44:30: error: no matching function for call to
 sqlpp::statement_t<void, sqlpp::select_t, sqlpp::no_select_flag_list_t,
sqlpp::select_column_list_t<void, test::TabBar_::Alpha>, sqlpp::no_from_t, sqlpp::no_extra_tables_t,
sqlpp::no_where_t<true>, sqlpp::no_group_by_t, sqlpp::no_having_t, sqlpp::no_order_by_t,
sqlpp::no_limit_t, sqlpp::no_offset_t>::from(test::TabBar_::Alpha&)
  select(t.alpha).from(t.alpha);

tests/SelectTest.cpp:44:30: note: candidate is:
In file included from include/sqlpp11/select.h:36:0,
                 from tests/SelectTest.cpp:31:
include/sqlpp11/from.h:178:11: note: template<class ... Tables>
typename std::enable_if<std::integral_constant<bool,
std::is_same<sqlpp::detail::logic_helper<sqlpp::detail::is_table_impl<Tables, void>::type::value ...>,
sqlpp::detail::logic_helper<true || (sqlpp::detail::is_table_impl<Tables, void>::type::value)...>
 >::value>::value, typename Clauses::_new_statement_t<sqlpp::no_from_t, sqlpp::from_t<void,
 Tables ...> > >::type sqlpp::no_from_t::_methods_t<Clauses>::from(Tables ...) const [with Tables =
{Tables ...}; Clauses = sqlpp::detail::statement_policies_t<void, sqlpp::select_t,
sqlpp::no_select_flag_list_t, sqlpp::select_column_list_t<void, test::TabBar_::Alpha>, sqlpp::no_from_t,
sqlpp::no_extra_tables_t, sqlpp::no_where_t<true>, sqlpp::no_group_by_t, sqlpp::no_having_t,
sqlpp::no_order_by_t, sqlpp::no_limit_t, sqlpp::no_offset_t>]
      auto from(Tables... tables) const

include/sqlpp11/from.h:178:11: note:  template argument
deduction/substitution failed:
include/sqlpp11/from.h: In substitution of  template<class ...
Tables> typename std::enable_if<std::integral_constant<bool,
std::is_same<sqlpp::detail::logic_helper<sqlpp::detail::is_table_impl<Tables, void>::type::value ...>,
sqlpp::detail::logic_helper<true || (sqlpp::detail::is_table_impl<Tables, void>::type::value)...>
>::value>::value, typename Clauses::_new_statement_t<sqlpp::no_from_t, sqlpp::from_t<void,
```

Recap

Compilers make a difference. . .

. . . so you should support compiler developers.

Let's try ~~enable_if and~~ Concepts Lite

Using Concepts Lite

```
template<typename T>
concept bool Table()
{
  return is_table_t<T>::value;
}
```

Let's try ~~enable_if and~~ Concepts Lite

Using Concepts Lite

```
template<typename T>
concept bool Table()
{
  return is_table_t<T>::value;
}

template<Table... Tables>
auto from(Tables... tables) const
-> _new_statement_t<from_t<void, Tables...>>
{
    //...
}
```

Let's try ~~enable_if and~~ Concepts Lite

gcc-5.0's error messages

```
In file included from include/sqlpp11/select.h:35:0,
                 from tests/SelectTest.cpp:31:
include/sqlpp11/from.h: In instantiation of  struct sqlpp::no_from_t::_methods_t<sqlpp::detail::statement_po
sqlpp::select_t, sqlpp::no_select_flag_list_t, sqlpp::no_select_column_list_t, sqlpp::no_from_t, sqlpp::no_ext
sqlpp::no_where_t<true>, sqlpp::no_group_by_t, sqlpp::no_having_t, sqlpp::no_order_by_t, sqlpp::no_limit_t, sq
include/sqlpp11/statement.h:123:11:   required from  struct sqlpp::statement_t<void, sqlpp::select_t,
sqlpp::no_select_flag_list_t, sqlpp::no_select_column_list_t, sqlpp::no_from_t, sqlpp::no_extra_tables_t,
sqlpp::no_where_t<true>, sqlpp::no_group_by_t, sqlpp::no_having_t, sqlpp::no_order_by_t, sqlpp::no_limit_t, sq
include/sqlpp11/select.h:82:37:   required from here
include/sqlpp11/from.h:176:11: error: invalid use of pack expansion expression
      auto from(Tables... tables)
...
```

Tons of erroneous errors.

Let's try ~~enable_if and~~ Concepts Lite

Using Concepts Lite

```
template<typename... Tables>
requires all_t<is_table_t<Tables>::value...>::value
auto from(Tables... tables)
-> _new_statement_t<from_t<void, Tables...>>
{
  //...
}
```

Let's try ~~enable_if and~~ Concepts Lite

gcc-5.0's error messages

```
tests/SelectTest.cpp:44:52: error: no matching function for call to  s q l p p::state
sqlpp::no_offset_t>::from(const char [2])
  for (const auto& row : db(select(t.alpha).from("t").where(true)))
In file included from include/sqlpp11/select.h:35:0,
                 from tests/SelectTest.cpp:31:
include/sqlpp11/from.h:176:11: note: candidate: sqlpp::no_from_t::_methods_t<Clause
      auto from(Tables... tables)

include/sqlpp11/from.h:176:11: note:   constraints not satisfied
include/sqlpp11/from.h:176:11: note:     all_t<sqlpp::detail::is_table_impl<Tables
                                                   evaluated to false
```

Let's try ~~enable_if and~~ Concepts Lite

gcc-5.0's error messages if it weren't for the bug

```
tests/SelectTest.cpp:44:52: error: no matching function for call to  s q l p p ::state
sqlpp::no_offset_t>::from(const char [2])
  for (const auto& row : db(select(t.alpha).from("t").where(true)))

In file included from include/sqlpp11/select.h:35:0,
                 from tests/SelectTest.cpp:31:
include/sqlpp11/from.h:176:11: note: candidate: sqlpp::no_from_t::_methods_t<Clause
      auto from(Tables... tables)

include/sqlpp11/from.h:176:11: note:   constraints not satisfied
include/sqlpp11/from.h:176:11: note:   note:
in the expansion of    (Table<Tables>)()...
include/sqlpp11/from.h:176:11: note:   note:
   Table<const char*>()    was not satisfied
```

Let's try ~~enable_if and~~ Concepts Lite

gcc-5.0's error messages if it weren't for the bug

```
tests/SelectTest.cpp:44:52: error: no matching function for call to  sqlpp::state
sqlpp::no_offset_t>::from(const char [2])
  for (const auto& row : db(select(t.alpha).from("t").where(true)))

In file included from include/sqlpp11/select.h:35:0,
                 from tests/SelectTest.cpp:31:
include/sqlpp11/from.h:176:11: note: candidate: sqlpp::no_from_t::_methods_t<Clause
      auto from(Tables... tables)

include/sqlpp11/from.h:176:11: note:   constraints not satisfied
include/sqlpp11/from.h:176:11: note:   note:
in the expansion of    (Table<Tables>)()...
include/sqlpp11/from.h:176:11: note:   note:
   Table<const char*>()    was not satisfied
```

Not bad.

static_assert, enable_if and Concepts Lite

Well. . .

- static_assert
  - specific messages
  - compiler continues and spills
  - a static assert in a function body is too late if there are problems in the return type
- enable_if
  - ugly code
  - gcc gives ugly error messages
- Concepts Lite
  - nice code
  - halfway decent error messages
  - not ready yet (your input is needed)
  - you require aptly named concepts for everything

static_assert, enable_if and Concepts Lite

Well. . .

- static_assert
  - specific messages
  - compiler continues and spills
  - a static assert in a function body is too late if there are problems in the return type
- enable_if
  - ugly code
  - gcc gives ugly error messages
- Concepts Lite
  - nice code
  - halfway decent error messages
  - not ready yet (your input is needed)
  - you require aptly named concepts for everything

Let's combine methods.

Combined efforts

Using enable_if *and* static_assert

```
template<typename... Tables>
auto from(Tables... tables) const
-> typename std::enable_if<
      all_t<is_table_t<Tables>::value...>::value,
      _new_statement_t<from_t<void, Tables...>>
      >::type;
```

Combined efforts

Using enable_if *and* static_assert

```
template<typename... Tables>
auto from(Tables... tables) const
-> typename std::enable_if<
      all_t<is_table_t<Tables>::value...>::value,
      _new_statement_t<from_t<void, Tables...>>
      >::type;

template<typename... Tables>
auto from(Tables... tables) const
-> typename std::enable_if<
      not all_t<is_table_t<Tables>::value...>::value,
      void
      >::type
{
    static_assert(wrong_t<Tables...>::value,
        "At least one argument of from is not a table");
}
```

Combined efforts

Using enable_if *and* static_assert

```
tests/SelectTest.cpp:43:53: error: member reference base type
      'typename std::enable_if<!all_t<is_table_t<const char *>::value>::value, voi
      (aka 'void') is not a structure or union
        for (const auto& row : db(select(t.alpha).from("t").where(true)))
```

Combined efforts

Using enable_if *and* static_assert

```
tests/SelectTest.cpp:43:53: error: member reference base type
      'typename std::enable_if<!all_t<is_table_t<const char *>::value>::value, voic
      (aka 'void') is not a structure or union
        for (const auto& row : db(select(t.alpha).from("t").where(true)))

In file included from tests/SelectTest.cpp:31:
In file included from include/sqlpp11/select.h:35:
include/sqlpp11/from.h:190:10: error: static_assert failed "At least one argument c
            static_assert(wrong_t<Tables...>::value, "At least one argument of fron

tests/SelectTest.cpp:43:44: note: in instantiation of function template specializat
      sqlpp::no_select_flag_list_t, sqlpp::select_column_list_t<void, sqlpp::columr
      sqlpp::no_group_by_t, sqlpp::no_having_t, sqlpp::no_order_by_t, sqlpp::no_lir
      sqlpp::no_offset_t> >::from<const char *>' requested here
        for (const auto& row : db(select(t.alpha).from("t").where(true)))
```

Combined efforts

Using enable_if *and* static_assert

```
tests/SelectTest.cpp:43:53: error: member reference base type
      'typename std::enable_if<!all_t<is_table_t<const char *>::value>::value, voic
      (aka 'void') is not a structure or union
        for (const auto& row : db(select(t.alpha).from("t").where(true)))


In file included from tests/SelectTest.cpp:31:
In file included from include/sqlpp11/select.h:35:
include/sqlpp11/from.h:190:10: error: static_assert failed "At least one argument c
        static_assert(wrong_t<Tables...>::value, "At least one argument of fron

tests/SelectTest.cpp:43:44: note: in instantiation of function template specializat
      sqlpp::no_select_flag_list_t, sqlpp::select_column_list_t<void, sqlpp::columr
      sqlpp::no_group_by_t, sqlpp::no_having_t, sqlpp::no_order_by_t, sqlpp::no_lir
      sqlpp::no_offset_t> >::from<const char *>' requested here
        for (const auto& row : db(select(t.alpha).from("t").where(true)))
```

Nice, but you need one additional ”error-overload” for each function (API bloat).

Combined efforts

Using enable_if *and* static_assert

```
tests/SelectTest.cpp:43:53: error: member reference base type
      'typename std::enable_if<!all_t<is_table_t<const char *>::value>::value, voi
      (aka 'void') is not a structure or union
        for (const auto& row : db(select(t.alpha).from("t").where(true)))

In file included from tests/SelectTest.cpp:31:
In file included from include/sqlpp11/select.h:35:
include/sqlpp11/from.h:190:10: error: static_assert failed "At least one argument o
          static_assert(wrong_t<Tables...>::value, "At least one argument of fro

tests/SelectTest.cpp:43:44: note: in instantiation of function template specializa
      sqlpp::no_select_flag_list_t, sqlpp::select_column_list_t<void, sqlpp::colum
      sqlpp::no_group_by_t, sqlpp::no_having_t, sqlpp::no_order_by_t, sqlpp::no_li
      sqlpp::no_offset_t> >::from<const char *>' requested here
        for (const auto& row : db(select(t.alpha).from("t").where(true)))
```

Nice, but you need one additional "error-overload" for each function (API bloat).
And all hell breaks loose if all overloads fail.

Combined efforts

static_assert and tag dispatch

Combined efforts

static_assert and tag dispatch

```
template<typename... Tables>
auto from(Tables... tables) const
-> decltype(_from_impl(all_t<is_table_t<Tables>::value...>{}, tables...))
{
  static_assert(all_t<is_table_t<Tables>::value...>::value,
        "at least one argument is not a table or join in from()");
```

Combined efforts

static_assert and tag dispatch

```
template<typename... Tables>
auto from(Tables... tables) const
-> decltype(_from_impl(all_t<is_table_t<Tables>::value...>{}, tables...))
{
  static_assert(all_t<is_table_t<Tables>::value...>::value,
        "at least one argument is not a table or join in from()");

  return _from_impl(all_t<is_table_t<Tables>::value...>{}, tables...);
}
```

Combined efforts

static_assert and tag dispatch

```
template<typename... Tables>
auto from(Tables... tables) const
-> decltype(_from_impl(all_t<is_table_t<Tables>::value...>{}, tables...))
{
  static_assert(all_t<is_table_t<Tables>::value...>::value,
        "at least one argument is not a table or join in from()");

  return _from_impl(all_t<is_table_t<Tables>::value...>{}, tables...);
}

template<typename... Tables>
auto _from_impl(const std::true_type&, Tables... tables) const
{
    //...
}
```

Combined efforts

static_assert and tag dispatch

```
template<typename... Tables>
auto from(Tables... tables) const
-> decltype(_from_impl(all_t<is_table_t<Tables>::value...>{}, tables...))
{
  static_assert(all_t<is_table_t<Tables>::value...>::value,
        "at least one argument is not a table or join in from()");

  return _from_impl(all_t<is_table_t<Tables>::value...>{}, tables...);
}

template<typename... Tables>
auto _from_impl(const std::true_type&, Tables... tables) const
{
    //...
}

template<typename... Tables>
auto _from_impl(const std::false_type&, Tables... tables) const
-> bad_statement;
```

Combined efforts

static_assert and tag dispatch

```
tests/Select.cpp:52:54: error: no member named 'where' in 'sqlpp::bad_statement'
        for (const auto& row : db(select(t.alpha).from("t").where(true)))

In file included from tests/Select.cpp:31:
In file included from include/sqlpp11/select.h:37:
include/sqlpp11/from.h:170:7: error: static_assert failed "at least one argument is
            static_assert(_check<Tables...>::value, "at least one argument is not a

tests/Select.cpp:52:44: note: in instantiation of function template specialization
      sqlpp::select_t, sqlpp::no_select_flag_list_t, sqlpp::select_column_list_t<vo
      sqlpp::no_group_by_t, sqlpp::no_having_t, sqlpp::no_order_by_t, sqlpp::no_lin
        for (const auto& row : db(select(t.alpha).from("t").where(true)))
```

Combined efforts

Nice. We reduced hundreds of lines of useless error messages to three lines with a clear message.

Combined efforts

Nice. We reduced hundreds of lines of useless error messages to three lines with a clear message.

But . . .

- I heard that static_assert is not SFINAE-friendly?

Combined efforts

Nice. We reduced hundreds of lines of useless error messages to three lines with a clear message.

But . . .

- I heard that static_assert is not SFINAE-friendly?
    - It is a hard error.
    - If it is in an unevaluated body, it will not even fire.

static_assert is not SFINAE-friendly

For SFINAE either use

- your function's return type or
- your documented check

```
template<typename... Tables>
auto from(Tables... tables) const
-> decltype(_from_impl(all_t<is_table_t<Tables>::value...>{}, tables...))
{
  static_assert(all_t<is_table_t<Tables>::value...>::value, "at least one argument

  return _from_impl(all_t<is_table_t<Tables>::value...>, tables...);
}
```

static_assert is not SFINAE-friendly

For SFINAE either use

- your function's return type or
- your documented check

```
template<typename... Tables>
auto from(Tables... tables) const
-> decltype(_from_impl(all_t<is_table_t<Tables>::value...>{}, tables...))
{
  static_assert(all_t<is_table_t<Tables>::value...>::value, "at least one argument

  return _from_impl(all_t<is_table_t<Tables>::value...>, tables...);
}
```

And do *NOT* use C++14 return type deduction!

static_assert is not SFINAE-friendly

For SFINAE either use

- your function's return type or
- your documented check

```
template<typename... Tables>
auto from(Tables... tables) const
-> decltype(_from_impl(all_t<is_table_t<Tables>::value...>{}, tables...))
{
  static_assert(all_t<is_table_t<Tables>::value...>::value, "at least one argument

  return _from_impl(all_t<is_table_t<Tables>::value...>, tables...);
}
```

And do *NOT* use C++14 return type deduction!
It would cause the compiler to run into the static_assert in the function body.

Nice. We reduced hundreds of lines of useless error messages to three lines with a clear message.
But . . .

- I heard that static_assert is not SFINAE-friendly?
- what if I have several different (even unknown) checks to perform?

Wait a second.

Wait a second.

Unknown conditions?

Several "unknown" conditions

Several "unknown" conditions

A harmless piece of code...

```
struct MockDb
{
    //...
    template<typename Sql>
    auto operator() (const Sql& statement)
    -> decltype(statement._run(*this))
    {
        return statement._run(*this);
    }
    //...
};
```

Several "unknown" conditions

A harmless piece of code. . .

```
struct MockDb
{
    //...
    template<typename Sql>
    auto operator() (const Sql& statement)
    -> decltype(statement._run(*this))
    {
        return statement._run(*this);
    }
    //...
};
```

. . . Called by an unsuspecting developer. . .

```
db(select(t.alpha).from(t));
```

Several "unknown" conditions

. . . produces these error messages

```
In file included from tests/SelectTest.cpp:31:
In file included from include/sqlpp11/select.h:38:
include/sqlpp11/where.h:230:7: error: static_assert failed
        "where expression required, e.g. where(true)" static_assert(not _required, "where expression required,
```

Several "unknown" conditions

. . . produces these error messages

```
In file included from tests/SelectTest.cpp:31:
In file included from include/sqlpp11/select.h:38:
include/sqlpp11/where.h:230:7: error: static_assert failed
        "where expression required, e.g. where(true)" static_assert(not _required, "where expression required,


include/sqlpp11/statement.h:189:98: note: in instantiation of member function
      'sqlpp::no_where_t<true>::_methods_t<sqlpp::detail::statement_policies_t<void, sqlpp::select_t, sqlpp::n
requested here
       (void) swallow{(Clauses::template _methods_t<detail::statement_policies_t<Db, Clauses...>>
       ::_check_consistency(), 0)...};
```

Several "unknown" conditions

. . . produces these error messages

```
In file included from tests/SelectTest.cpp:31:
In file included from include/sqlpp11/select.h:38:
include/sqlpp11/where.h:230:7: error: static_assert failed
        "where expression required, e.g. where(true)" static_assert(not _required, "where expression required,


include/sqlpp11/statement.h:189:98: note: in instantiation of member function
      'sqlpp::no_where_t<true>::_methods_t<sqlpp::detail::statement_policies_t<void, sqlpp::select_t, sqlpp::n
requested here
       (void) swallow{(Clauses::template _methods_t<detail::statement_policies_t<Db, Clauses...>>
       ::_check_consistency(), 0)...};


include/sqlpp11/select_column_list.h:310:22: note: in instantiation of member function 'sqlpp::statement_t<voi
                         _statement_t::_check_consistency();
```

Several "unknown" conditions

. . . produces these error messages

```
In file included from tests/SelectTest.cpp:31:
In file included from include/sqlpp11/select.h:38:
include/sqlpp11/where.h:230:7: error: static_assert failed
        "where expression required, e.g. where(true)" static_assert(not _required, "where expression required,


include/sqlpp11/statement.h:189:98: note: in instantiation of member function
      'sqlpp::no_where_t<true>::_methods_t<sqlpp::detail::statement_policies_t<void, sqlpp::select_t, sqlpp::n
requested here
       (void) swallow{(Clauses::template _methods_t<detail::statement_policies_t<Db, Clauses...>>
       ::_check_consistency(), 0)...};


include/sqlpp11/select_column_list.h:310:22: note: in instantiation of member function 'sqlpp::statement_t<voi
                         _statement_t::_check_consistency();


include/sqlpp11/statement.h:195:43: note: in instantiation of function template specialization 'sqlpp::select_
      sqlpp::no_where_t<true>, sqlpp::no_group_by_t, sqlpp::no_having_t, sqlpp::no_order_by_t, sqlpp::no_limit
      >' requested here
                        return _result_methods_t<statement_t>::_run(db);
```

Several "unknown" conditions

. . . produces these error messages

```
In file included from tests/SelectTest.cpp:31:
In file included from include/sqlpp11/select.h:38:
include/sqlpp11/where.h:230:7: error: static_assert failed
        "where expression required, e.g. where(true)" static_assert(not _required, "where expression required,


include/sqlpp11/statement.h:189:98: note: in instantiation of member function
      'sqlpp::no_where_t<true>::_methods_t<sqlpp::detail::statement_policies_t<void, sqlpp::select_t, sqlpp::n
requested here
       (void) swallow{(Clauses::template _methods_t<detail::statement_policies_t<Db, Clauses...>>
       ::_check_consistency(), 0)...};


include/sqlpp11/select_column_list.h:310:22: note: in instantiation of member function 'sqlpp::statement_t<voi
                         _statement_t::_check_consistency();


include/sqlpp11/statement.h:195:43: note: in instantiation of function template specialization 'sqlpp::select_
      sqlpp::no_where_t<true>, sqlpp::no_group_by_t, sqlpp::no_having_t, sqlpp::no_order_by_t, sqlpp::no_limit
      >' requested here
                        return _result_methods_t<statement_t>::_run(db);


tests/MockDb.h:112:13: note: in instantiation of function template specialization 'sqlpp::statement_t<void, sq
      sqlpp::no_offset_t>::_run<MockDb<false> >' requested here
                        return statement._run(*this);
```

Several "unknown" conditions

...produces these error messages

```
In file included from tests/SelectTest.cpp:31:
In file included from include/sqlpp11/select.h:38:
include/sqlpp11/where.h:230:7: error: static_assert failed
        "where expression required, e.g. where(true)" static_assert(not _required, "where expression required,


include/sqlpp11/statement.h:189:98: note: in instantiation of member function
      'sqlpp::no_where_t<true>::_methods_t<sqlpp::detail::statement_policies_t<void, sqlpp::select_t, sqlpp::n
requested here
       (void) swallow{(Clauses::template _methods_t<detail::statement_policies_t<Db, Clauses...>>
       ::_check_consistency(), 0)...};


include/sqlpp11/select_column_list.h:310:22: note: in instantiation of member function 'sqlpp::statement_t<voi
                         _statement_t::_check_consistency();


include/sqlpp11/statement.h:195:43: note: in instantiation of function template specialization 'sqlpp::select_
      sqlpp::no_where_t<true>, sqlpp::no_group_by_t, sqlpp::no_having_t, sqlpp::no_order_by_t, sqlpp::no_limit
      >' requested here
                        return _result_methods_t<statement_t>::_run(db);


tests/MockDb.h:112:13: note: in instantiation of function template specialization 'sqlpp::statement_t<void, sq
      sqlpp::no_offset_t>::_run<MockDb<false> >' requested here
                        return statement._run(*this);


tests/SelectTest.cpp:44:4: note: in instantiation of function template specialization 'MockDb<false>::operator
        db(sqlpp::select(t.alpha).from(t));
```

Several "unknown" conditions

How do we get the argument check here?

```
struct MockDb
{
    //...
    template<typename Sql>
    auto operator() (const Sql& statement)
    -> decltype(statement._run(*this))
    {
        return statement._run(*this);
    }
    //...
};
```

Let's looks at an sqlpp11 statement.

Several "unknown" conditions

sqlpp11 statement

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

Several "unknown" conditions

sqlpp11 statement

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

The `Db::operator()` has no way of knowing what to check in all detail, it seems.

Combine check results

Move check results from deeper in the hierarchy to the API border.

```
template<typename Database, typename... Clauses>
struct statement
{
    //...
    using run_check = all_t<Clauses::run_check::value...>;
};
```

Use the aggregated check

Aggregated check plus tag dispatch

Use the aggregated check

Aggregated check plus tag dispatch

```
struct MockDb
{
    //...
    template<typename Sql>
    auto operator() (const Sql& statement)
    -> decltype(this->_run(statement, typename Sql::run_check_t{}))
    {
        static_assert(Sql::_run_check::value, "statement is inconsistent");
        return _run(statement, typename Sql::_run_check{});
    }
```

Use the aggregated check

Aggregated check plus tag dispatch

```
struct MockDb
{
    //...
    template<typename Sql>
    auto operator() (const Sql& statement)
    -> decltype(this->_run(statement, typename Sql::run_check_t{}))
    {
        static_assert(Sql::_run_check::value, "statement is inconsistent");
        return _run(statement, typename Sql::_run_check{});
    }

    template<typename Sql>
    auto _run(const Sql& statement, const std::true_type&) -> decltype(statement._r
    {
        return statement._run(*this);
    }

    template<typename Sql>
    auto _run(const Sql& statement, const std::false_type&) -> void;
    //...
};
```

Use the aggregated check

Short error message

```
In file included from tests/SelectTest.cpp:28:
tests/MockDb.h:112:4: error: static_assert failed "statement is inconsistent"
            static_assert(Sql::_run_check::value, "statement is inconsistent");

tests/SelectTest.cpp:44:4: note: in instantiation of function template specializati
        db(sqlpp::select(t.alpha).from(t));
```

Use the aggregated check

Short error message

```
In file included from tests/SelectTest.cpp:28:
tests/MockDb.h:112:4: error: static_assert failed "statement is inconsistent"
            static_assert(Sql::_run_check::value, "statement is inconsistent");

tests/SelectTest.cpp:44:4: note: in instantiation of function template specializati
        db(sqlpp::select(t.alpha).from(t));
```

Short, but also not very helpful.

Use the aggregated check

Short error message

```
In file included from tests/SelectTest.cpp:28:
tests/MockDb.h:112:4: error: static_assert failed "statement is inconsistent"
            static_assert(Sql::_run_check::value, "statement is inconsistent");

tests/SelectTest.cpp:44:4: note: in instantiation of function template specializati
        db(sqlpp::select(t.alpha).from(t));
```

Short, but also not very helpful.
We need to get the specific message again (but without the clutter).

Portable static asserts

A portable static assert...

Portable static asserts

A portable static assert...

```
struct assert_where_t
{
    using type = std::false_type;
```

Portable static asserts

A portable static assert. . .

```
struct assert_where_t
{
    using type = std::false_type;

    template<typename T = void>
    static void _()
    {
        static_assert(wrong_t<T>::value,
              "where expression required, e.g. where(true)");
    };
};
```

Portable static asserts

A portable static assert...

```
struct assert_where_t
{
    using type = std::false_type;

    template<typename T = void>
    static void _()
    {
        static_assert(wrong_t<T>::value,
                "where expression required, e.g. where(true)");
    };
};
```

...and an OK type

```
struct consistent_t
{
    using type = std::true_type;
    static void _() {};
};
```

Portable static asserts

Move check results from deeper in the hierarchy to the API border.

```
template<typename Database, typename... Clauses>
struct statement
{
    //...
    using _run_check = detail::get_first_if<is_inconsistent_t, consistent_t,
          typename Clauses::_consistency_check...>;

    using _run_check_t = typename _run_check::type;
    //...
};
```

Portable static asserts

Call the portable static_assert at the API border

```
struct MockDb
{
    //...
    template<typename Sql>
    auto operator() (const Sql& statement)
    -> decltype(this->_run(statement, typename Sql::run_check_t{}))
    {
        Sql::_run_check::_();
        return _run(statement, typename Sql::_run_check_t{});
    }
    //...
};
```

Portable static asserts

Short and helpful error message

```
include/sqlpp11/where.h:191:5: error: static_assert failed
                       "where expression required, e.g. where(true)"
    static_assert(wrong_t<T>::value, "where expression required, e.g. where(true)")

tests/MockDb.h:124:19: note: in instantiation of function template specialization
                        Sql::_run_check::_();

tests/SelectTest.cpp:44:4: note: in instantiation of function template specializat
        db(select(t.alpha).from(t));
```

Short *and* helpful.

Final touch

Are we there yet?

Final touch

Are we there yet?

```
struct MockDb
{
    //...
    template<typename Sql>
    auto operator() (const Sql& statement)
    -> decltype(this->_run(statement, typename Sql::run_check_t{}))
    {
        Sql::_run_check::_(); // SUCH CHECKS MIGHT CAUSE ERRORS
        return _run(statement, typename Sql::_run_check{});
    }
    //...
};
```

Final touch

Wrap the statement's _run_check

```
namespace detail
{
  template <typename T, typename Enable = void>
  struct run_check
  {
    using type = assert_run_statement_or_prepared_t;
  };
```

Final touch

Wrap the statement's _run_check

```
namespace detail
{
  template <typename T, typename Enable = void>
  struct run_check
  {
    using type = assert_run_statement_or_prepared_t;
  };

  template <typename T>
  struct run_check<T, typename std::enable_if<is_statement_t<T>::value>::type>
  {
    using type = T::_run_check;
  };
}
```

Final touch

Wrap the statement's _run_check

```
namespace detail
{
  template <typename T, typename Enable = void>
  struct run_check
  {
    using type = assert_run_statement_or_prepared_t;
  };

  template <typename T>
  struct run_check<T, typename std::enable_if<is_statement_t<T>::value>::type>
  {
    using type = T::_run_check;
  };
}

template <typename T>
using run_check_t = typename detail::run_check<T>::type;
```

Final touch

Wrap the statement's _run_check

```
struct MockDb
{
    //...
    template<typename Sql>
    auto operator() (const Sql& statement)
    -> decltype(this->_run(statement, sqlpp::run_check_t<Sql>{})) // harmless
    {
        sqlpp::run_check_t<Sql>::_(); // also asserts Sql is an sqlpp::statement
        return _run(statement, sqlpp::run_check_t<Sql>{});
    }
    //...
};
```

Final touch

Try to execute a string

```
include/sqlpp11/type_traits.h:292:7: error: static_assert failed
      "connection cannot run something that is neither statement nor prepared state
      static_assert(wrong_t<T>::value,

tests/MockDb.h:123:51: note: in instantiation of function template specialization
    sqlpp::run_check_t<Sql>::_();

tests/Select.cpp:53:5: note: in instantiation of function template specialization
  db("select * from tab");
```

Final touch

Try to execute a string

```
include/sqlpp11/type_traits.h:292:7: error: static_assert failed
      "connection cannot run something that is neither statement nor prepared state
      static_assert(wrong_t<T>::value,

tests/MockDb.h:123:51: note: in instantiation of function template specialization
    sqlpp::run_check_t<Sql>::_();

tests/Select.cpp:53:5: note: in instantiation of function template specialization
  db("select * from tab");
```

OK, now we're done.

Summary

My suggestions for reducing template error messages

- Support compiler developers.
- Support library authors.
- Try to minimize the exposed templates.
- Try to minimize recursion.
- Perform parameter validation at the API border.
- Document your validation conditions.
- Use static_assert *with great care*
    - Avoid static asserts in return types.
    - Use tag dispatch.
    - Use portable static asserts.
- Experiment with Concepts Lite!

Thank You!

Questions?