How to test static_assert

Dr. Roland Bock

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

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

CppCon 2016, 2016-09-22

An UPDATE example using sqlpp11, an EDSL for SQL in C++

```
const auto count = db(update(tab).set(tab.name = "John Doe"));
```

Compiled with gcc/clang:

```
sqlpp11/test.cpp:136:10: error: variable has incomplete type 'void'
    const auto count = db.prepare(update(t).set(t.beta = ""));
               ^
In file included from /home/rbock/projects/sqlpp11/test.cpp:30:
In file included from /home/rbock/projects/sqlpp11/include/sqlpp11/where.h:38:
sqlpp11/include/sqlpp11/where.h:214:3: error: static_assert failed
        "calling where() or unconditionally() required"
```

static_assert is cool, it

- tests conditions at compile time,
- produces a hard compile error,
- yields hand-written error messages.

But it is only cool, if you get both right, the condition and the message!

To make things more complicated. . .

To make things more complicated...

https://github.com/rbock/sqlpp11-connector-mysql/issues/19

To make things more complicated...

https://github.com/rbock/sqlpp11-connector-mysql/issues/19

Compiled with MSVC:

```
error C3313: 'count': variable cannot have the type 'void'
```

To make things more complicated...

https://github.com/rbock/sqlpp11-connector-mysql/issues/19

Compiled with MSVC:

```
error C3313: 'count': variable cannot have the type 'void'
```

My precious static_assert is gone!

[intro.compliance]

If a program contains a violation of any diagnosable rule or [...], a conforming implementation shall issue *at least one diagnostic message*.

static_assert is cool, it

- tests conditions at compile time,
- produces a hard compile error,
- yields hand-written error messages.

static_assert is cool, it

- tests conditions at compile time,
- produces a hard compile error,
- yields hand-written error messages.

But

- the assert conditions might be wrong,
- sometimes it goes missing,
- **and it needs to be tested!**

Goal:

I want to assert that all my static_asserts actually fire exactly when they should.

Goal:

I want to assert that all my static_asserts actually fire exactly when they should.

And I also want to help MSVC users in case the static_assert gets lost.

Basic approaches

Code staring.

How about this?

```
#ifdef TEST_STATIC_ASSERT
    #define static_assert(A, B) assert((B, A))
#endif
```

Or this?

```
#ifdef TEST_STATIC_ASSERT
    #define static_assert(A, B) if (A) throw myAssertException(B);
#endif
```

`static_assert` is often used for code that cannot compile anyway.
Thus, replacing it with a runtime construct will simply not work.

`static_assert` is often used for code that cannot compile anyway.

Thus, replacing it with a runtime construct will simply not work.

What then?

Well, it is a compile error.

Well, it is a compile error.

Test it by compiling code.

CMake example

```
function(test_constraint name pattern)
  add_executable(${name} EXCLUDE_FROM_ALL ${name}.cpp)
  add_test(NAME ${name}
    COMMAND ${CMAKE_COMMAND} --build ${CMAKE_BINARY_DIR} --target ${name}
    )
  set_property(TEST ${name} PROPERTY PASS_REGULAR_EXPRESSION ${pattern})
endfunction()

test_constraint(max_of_max "max\\(\\) cannot be used on an aggregate function")
```

Triggering a static_assert

```
#include "Sample.h"
#include "MockDb.h"
#include <sqlpp11/functions.h>
#include <iostream>

MockDb db;

int main()
{
  const auto t = test::TabBar{};

  max(max(t.alpha));
}
```

Using the build system works fine, although

- it requires lots of boilerplate code
- pattern matching is no fun

Compiler developers need to test static_assert. Maybe they have tools?

Compiler developers need to test static_assert. Maybe they have tools?

Sure, for example, you can test clang diagnostic messages

Annotate code with expected diagnostics

```
int f(); // expected-note {{declared here}}

static_assert(f(), "f"); // expected-error {{static_assert expression is not an integral constant expres
```

Compile with:

```
/usr/local/clang_trunk/bin/clang++ -cc1 -fsyntax-only -verify
diagnostics.cpp -std=c++11
```

Clang-format is hurting tests

```
int f(); // expected-note {{declared here}}

static_assert(f(), "f");  // expected-error {{static_assert expression is not an
                          // integral constant expression}}, expected-note
                          // {{non-constexpr function 'f' cannot be used in a
                          // constant expression}}
```

Clang-format is hurting tests

```
error: 'error' diagnostics seen but not expected:
  File diagnostics.cpp Line 5: cannot find end ('}}') of expected string
  File diagnostics.cpp Line 6: cannot find start ('{{') of expected string
  File diagnostics.cpp Line 5: static_assert expression is not an integral constant expression
error: 'note' diagnostics seen but not expected:
  File diagnostics.cpp Line 5: non-constexpr function 'f' cannot be used in a constant expression
4 errors generated.
```

Analysing compiler diagnostics

- depends on the compiler
- depends on the compiler version
- breaks by using modern coding tools

Does any of this work in real life?

A simple example

```
struct A
{
  auto bar() -> void;
};

template<typename T>
auto foo(T t) -> void
{
  static_assert(std::is_base_of<A, T>::value,
                "Argument needs to be derived from A");
  t.bar();
}

int main()
{
  foo(7);
}
```

Compiling

```
simple_assert.cpp:11:2: error: static_assert failed "Argument needs to be derived from A"
        static_assert(std::is_base_of<A, T>::value, "Argument needs to be derived from A");
        ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
simple_assert.cpp:17:2: note: in instantiation of function template specialization 'foo<int>' requested
        foo(7);
        ^
simple_assert.cpp:12:3: error: member reference base type 'int' is not a structure or union
        t.bar();
        ~^~~~
2 errors generated.
```

Compiling

```
simple_assert.cpp:11:2: error: static_assert failed "Argument needs to be derived from A"
        static_assert(std::is_base_of<A, T>::value, "Argument needs to be derived from A");
        ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~

simple_assert.cpp:17:2: note: in instantiation of function template specialization 'foo<int>' requested
        foo(7);
        ^

simple_assert.cpp:12:3: error: member reference base type 'int' is not a structure or union
        t.bar();
        ~^~~~

2 errors generated.
```

Most compilers do not stop after hitting a static_assert.

Compiling

```
simple_assert.cpp:11:2: error: static_assert failed "Argument needs to be derived from A"
        static_assert(std::is_base_of<A, T>::value, "Argument needs to be derived from A");
        ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
simple_assert.cpp:17:2: note: in instantiation of function template specialization 'foo<int>' requested
        foo(7);
        ^
simple_assert.cpp:12:3: error: member reference base type 'int' is not a structure or union
        t.bar();
        ~^~~~
2 errors generated.
```

Most compilers do not stop after hitting a static_assert.

Most compilers display the static_assert's message twice.

How do you handle multiple tests?

```
int main()
{
  foo(7);
  foo("cheesecake");
  foo(std::string{"whatever"});
  foo(A{});
}
```

How do you handle multiple tests?

```
int main()
{
  foo(7);
  foo("cheesecake");
  foo(std::string{"whatever"});
  foo(A{});
}
```

Trying to get this right with either build-system based tests or clang diagnostic tests is more effort than I would be willing to invest.

How about multiple static_assert? Example from sqlpp11 (early 2015)

```
template<typename... Tables>
auto from(Tables... tables) const
-> _new_statement_t<_check<Tables...>, from_t<void, from_table_t<Tables>...>>
{
    static_assert(all<is_table<Tables>::value...>::value, "at least one argument is not a table or join
    static_assert(unique_table_names<Tables...>::value, "at least one duplicate table name detected in f
    //...
}
```

How about multiple static_assert? Example from sqlpp11 (early 2015)

```
template<typename... Tables>
auto from(Tables... tables) const
-> _new_statement_t<_check<Tables...>, from_t<void, from_table_t<Tables>...>>
{
    static_assert(all<is_table<Tables>::value...>::value, "at least one argument is not a table or join
    static_assert(unique_table_names<Tables...>::value, "at least one duplicate table name detected in f
    //...
}
```

Testing this using the build system or compiler internals can get ugly.

We need to do something else.

First, let's clean up the mess.

Tag dispatch

```
template<typename T>
auto foo_impl(T t, std::true_type) -> void
{
  t.bar();
}

template<typename T>
auto foo_impl(T t, std::false_type) -> void;
```

Tag dispatch

```
template<typename T>
auto foo_impl(T t, std::true_type) -> void
{
  t.bar();
}

template<typename T>
auto foo_impl(T t, std::false_type) -> void;

template<typename T>
auto foo(T t) -> void
{
  static_assert(std::is_base_of<A, T>::value, "Argument needs to be derived from A");
  return foo_impl(t, std::is_base_of<A, T>{});
}
```

Now, the compiler only reports the static_assert.

Template-argument dependend return type

```
template<typename T>
auto foo_impl(T t, std::true_type) -> Bar<T>
{
  t.bar();
}

template<typename T>
auto foo_impl(T t, std::false_type) -> Bar<T>;

template<typename T>
auto foo(T t) -> Bar<T>
{
  static_assert(std::is_base_of<A, T>::value, "Argument needs to be derived from A");
  return foo_impl(t, std::is_base_of<A, T>{});
}
```

If the return type does not compile, you will get all those error messages, too.

Change return type for bad conditions

```
struct bad_statement {};
```

Change return type for bad conditions

```
struct bad_statement {};

template<typename T>
auto foo_impl(T t, std::true_type) -> Bar<T>
{
  t.bar();
}

template<typename T>
auto foo_impl(T t, std::false_type) -> bad_statement;
```

Change return type for bad conditions

```
struct bad_statement {};

template<typename T>
auto foo_impl(T t, std::true_type) -> Bar<T>
{
  t.bar();
}

template<typename T>
auto foo_impl(T t, std::false_type) -> bad_statement;

template<typename T>
auto foo(T t) -> decltype(foo_impl(t, std::is_base_of<A, T>{}))
{
  static_assert(std::is_base_of<A, T>::value, "Argument needs to be derived from A");
  return foo_impl(t, std::is_base_of<A, T>{});
}
```

Let's use the code

```
int main()
{
    foo(7).xxx();
}
```

gcc/clang

```
bad_statement_type.cpp:38:10: error: no member named 'xxx' in 'bad_statement'
        foo(7).xxx();
        ~~~~~~~ ^
bad_statement_type.cpp:32:2: error: static_assert failed "Argument needs to be derived from A"
        static_assert(std::is_base_of<A, T>::value, "Argument needs to be derived from A");
        ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
bad_statement_type.cpp:38:2: note: in instantiation of function template specialization 'foo<int>' reque
        foo(7).xxx();
        ^
2 errors generated.
```

MSVC

```
test.cpp(38): error C2039: 'xxx': is not a member of 'bad_statement'
test.cpp(22): note: see declaration of 'bad_statement'
```

MSVC

```
test.cpp(38): error C2039: 'xxx': is not a member of 'bad_statement'
test.cpp(22): note: see declaration of 'bad_statement'
```

That's all.

MSVC ignores the static_assert here.

MSVC ignores the static_assert here.

And we still have no way of actually testing the static_assert inside our function.

Making static_assert testable.

Conceptually, what do we need to do?

Conceptually, what do we need to do?

- Test the condition of the static_assert.
- Verify that the static_assert is actually part of the respective function.

Plan:

Plan:

We separate the condition from the static_assert and wrap the static_assert into a type that is returned in case the condition is not fulfilled.

For this, we define a small helper:

The wrong type

```
template<typename T>
struct wrong : std::false_type
{
};
```

For this, we define a small helper:

The wrong type

```
template<typename T>
struct wrong : std::false_type
{
};
```

The value of this template struct is always false, but the compiler does not know that until use a specialization of it.

Wrap the static_assert into a type

```
struct assert_arg_is_derived_from_a
{
  template<typename T = void>
  assert_arg_is_derived_from_a()
  {
    static_assert(wrong<T>::value, "argument needs to be derived from A");
  }
};
```

Counterpart to wrapped static_asserts

```
struct ok
{
};
```

Implement the static_assert's check again

```
template<typename T>
using check_arg = std::conditional_t<std::is_base_of<A, T>::value,
                                     ok,
                                     assert_arg_is_derived_from_a>;
```

Implement the static_assert's check again

```
template<typename T>
using check_arg = std::conditional_t<std::is_base_of<A, T>::value,
                                     ok,
                                     assert_arg_is_derived_from_a>;
```

If the check fails, check_arg will be the wrapped static_assert.

Good and bad function impl versions

```
template<typename T>
auto foo_impl(ok, T t) -> void
{
  t.bar();
}
```

Good and bad function impl versions

```
template<typename T>
auto foo_impl(ok, T t) -> void
{
  t.bar();
}

template<typename Check, typename T>
auto foo_impl(Check, T t) -> Check; // wrapped assert
```

Good and bad function impl versions

```
template<typename T>
auto foo_impl(ok, T t) -> void
{
  t.bar();
}

template<typename Check, typename T>
auto foo_impl(Check, T t) -> Check; // wrapped assert

template<typename T>
auto foo(T t) -> decltype(foo_impl(check_arg<T>{}, t))
{
  return foo_impl(check_arg<T>{}, t);
}
```

Test it

```
int main()
{
  foo(7).xxx;
  foo(A{});
}
```

gcc/clang

```
test.cpp:66:9: error: no member named 'xxx' in 'assert_arg_is_derived_from_a'
        foo(7).xxx;
        ~~~~~~ ^
test.cpp:26:3: error: static_assert failed "argument needs to be derived from A"
                static_assert(wrong<T>::value, "argument needs to be derived from A");
                ^             ~~~~~~~~~~~~~~~
test.cpp:60:19: note: in instantiation of function template specialization 'assert_arg_is_derived_from_a
        return foo_impl(check_arg<T>{}, t);
                        ^
test.cpp:66:2: note: in instantiation of function template specialization 'foo<int>' requested here
        foo(7).xxx;
        ^
2 errors generated.
```

MSVC

```
test.cpp(66): error C2039: 'xxx': is not a member of 'assert_arg_is_derived_from_a'
test.cpp(43): note: see declaration of 'assert_arg_is_derived_from_a'
```

If you ignore the return type (MSVC)

```
test.cpp(26): error C2338: argument needs to be derived from A
test.cpp(47): note: see reference to function template instantiation 'assert_arg_is_derived_from_a::asse
test.cpp(52): note: see reference to function template instantiation 'Check foo<int>(T)' being compiled
        with
        [
            Check=assert_arg_is_derived_from_a,
            T=int
        ]
```

So all compilers behave in a reasonable way.

Let's look at the return type again:

```
template<typename T>
auto foo(T t) -> decltype(foo_impl(check_arg<T>{}, t))
{
  return foo_impl(check_arg<T>{}, t);
}
```

Let's look at the return type again:

```
template<typename T>
auto foo(T t) -> decltype(foo_impl(check_arg<T>{}, t))
{
  return foo_impl(check_arg<T>{}, t);
}
```

The static_assert is only firing when we call the function, not when we analyse the return type.

Now we can test!

Now we can test!

```
int main()
{
   static_assert(std::is_same<decltype(foo(A{})), void>::value, "");
   static_assert(std::is_same<decltype(foo(7)),
                 assert_arg_is_derived_from_a>::value, "");
}
```

By changing the way that we link the static_assert to the function, the static_assert is suddenly trivial to test.

Going back the the original example

```
const auto count = db(update(tab).set(tab.name = "John Doe"));
```

clang/gcc

```
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); \
      ^             ~~~~~~~~~~~~~~~~~~~~
MockDb.h:125:20: note: in instantiation of function template specialization
      'sqlpp::assert_where_or_unconditionally_called_t::assert_where_or_unconditionally_called_t<>' requ
    return _run(t, sqlpp::run_check_t<_serializer_context_t, T>{});
                   ^
Update.cpp:66:24: note: in instantiation of function template specialization 'MockDbT<false>::operator()
      sqlpp::single_table_t<void, test::TabBar>, sqlpp::update_list_t<void, sqlpp::assignment_t<sqlpp::c
      sqlpp::no_where_t<true> > >' requested here
  const auto count = db(update(tab).set(tab.name = "John Doe"));
```

MSVC

```
where.h(213): error C2338: calling where() or unconditionally() required [C:\projects\sqlpp11\build\test
MockDb.h(125): note: see reference to function template instantiation 'sqlpp::assert_where_or_conditio
Update.cpp(66): note: see reference to function template instantiation 'Check MockDbT<false>::operator (
          with
          [
              Check=sqlpp::assert_where_or_unconditionally_called_t,
              Db=void,
              Table=test::TabBar,
              T=sqlpp::statement_t<void,sqlpp::update_t,sqlpp::single_table_t<void,test::TabBar>,sqlpp::
          ]
```

Surprisingly, now I even get the static_assert with MSVC!

Summary

- Use tag dispatch and failure return types to reduce the error spew of the compiler.
- Split the conditional static_assert into a wrapped, unconditional static_assert and a conditional type that is either harmless or the wrapped static_assert.
- Use the conditional type for tag dispatch and as failure return type.
- Write straight-forward compile time unit tests for your static_asserts, even if you have multiple static_asserts for single function.
- Sprinkle in a few build-system-based tests to make sure that there is no fundamental problem with your static_assert mechanics.

Summary

- Use tag dispatch and failure return types to reduce the error spew of the compiler.
- Split the conditional static_assert into a wrapped, unconditional static_assert and a conditional type that is either harmless or the wrapped static_assert.
- Use the conditional type for tag dispatch and as failure return type.
- Write straight-forward compile time unit tests for your static_asserts, even if you have multiple static_asserts for single function.
- Sprinkle in a few build-system-based tests to make sure that there is no fundamental problem with your static_assert mechanics.

Have fun!

Questions or bonus material?

constexpr-if (probably C++17)

Tempting. . .

```
template<typename T>
auto foo(T t)
{
  using Check = check_arg<T>;
  if constexpr(Check::value)
  {
    t.bar();
  }
  else
  {
    return Check{};
  }
}
```

Tempting. . .

```
template<typename T>
auto foo(T t)
{
  using Check = check_arg<T>;
  if constexpr(Check::value)
  {
    t.bar();
  }
  else
  {
    return Check{};
  }
}
```

Utterly cool and expressive. . .

. . . but this fails to compile

```
decltype(foo(7));
```

. . . but this fails to compile

```
decltype(foo(7));
```

Thus, we cannot test the return type at compile time.

We need to fix that. . .

Change the assert-struct a bit

```
struct assert_base {};

struct arg_is_derived_from_a : assert_base
{
    static constexpr auto value = false;

    template <typename T = void>
    static auto _() -> void
    {
      static_assert(wrong<T>::value, text);
    }
}
```

Change the assert-struct a bit

```
struct assert_base {};

struct arg_is_derived_from_a : assert_base
{
    static constexpr auto value = false;

    template <typename T = void>
    static auto _() -> void
    {
      static_assert(wrong<T>::value, text);
    }
}
```

The static_assert does not lurk in the constructor but in a static member function.

Add a wrapper

```
template<typename Assert>
struct bad_statement
{
  bad_statement(Assert)
  {
    Assert::_();
  }
};
```

Add a wrapper

```
template<typename Assert>
struct bad_statement
{
  bad_statement(Assert)
  {
    Assert::_();
  }
};

template <typename T>
using make_return_type =
    std::conditional_t<std::is_base_of<assert_base, T>::value,
                       bad_statement<T>,
                       T>;
```

Working constexpr-if version

```
template<typename T>
auto foo_impl(T t)
{
  using Check = check_arg<T>;
  if constexpr (Check::value)
  {
    t.bar();
  }
  else
  {
    return Check{};
  }
}

template<typename T>
auto foo(T t) -> make_return_type<decltype(foo_impl(t))>
{
  return foo_impl(t);
}
```

Very neat!

- No functions overloads,
- The argument check needs to be called only once!

concepts lite (hopefully C++20)

Not exactly as envisioned...

```
template<typename T>
requires check_arg<T>::value
auto foo(T t) -> void
{
  t.bar();
}

template<typename T>
auto foo(T t) -> check_arg<T>
{
  return{};
}
```

Not exactly as envisioned...

```
template<typename T>
requires check_arg<T>::value
auto foo(T t) -> void
{
  t.bar();
}

template<typename T>
auto foo(T t) -> check_arg<T>
{
  return{};
}
```

The default overload produces the static_assert.

Of course, you can omit all the static_assert stuff, if you're happy with the concept error message by the compiler.

Of course, you can omit all the static_assert stuff, if you're happy with the concept error message by the compiler.
You only need to figure out how to test concept-lite-constrained functions ;-)

Questions?

Thank you!