SQLite + C++14

Докладчик: Евгений Захаров

Какие уже есть ORM для SQLite на C++?

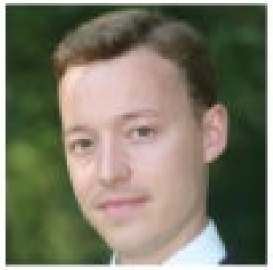
```
database db(«dbfile.db»);

db << "insert into user (age,name,weight) values (?,?,?);"
         << 20
         << u"bob"
         << 83.25;
```

SqliteModernCpp/**sqlite_modern_cpp**

https://github.com/SqliteModernCpp/sqlite_modern_cpp

Какие уже есть ORM для SQLite на C++?

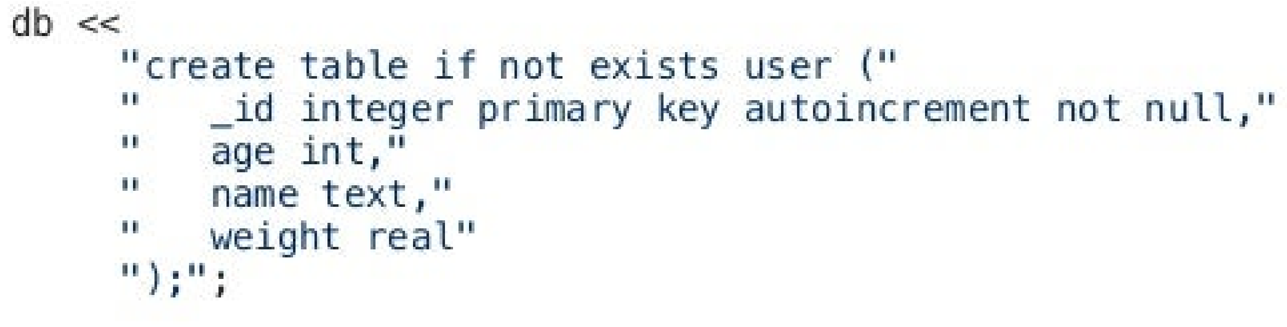
```
SQLite::Statement query(db, "SELECT id as test_id, value as test_val, weight as
test_weight FROM test WHERE weight > ?»);

query.bind(1, 2);

while (query.executeStep()) {

    const int id = query.getColumn(0);

    const std::string value = query.getColumn(1);

    const int bytes = query.getColumn(1).size();

    const double weight = query.getColumn(2);

}
```



SRombauts/SQLiteCpp

https://github.com/SRombauts/SQLiteCpp

Какие уже есть ORM для SQLite на C++?



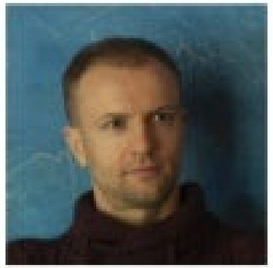
```
db <<
     "create table if not exists user ("
     "   _id integer primary key autoincrement not null,"
     "   age int,"
     "   name text,"
     "   weight real"
     ");";
```

SqliteModernCpp/**sqlite_modern_cpp**

https://github.com/SqliteModernCpp/sqlite_modern_cpp

Какие уже есть ORM для SQLite на C++?

```
class Person{
    friend class hiberlite::access;

    template<class Archive>
    void hibernate(Archive & ar)
    {
        ar & HIBERLITE_NVP(name);
        ar & HIBERLITE_NVP(age);
        ar & HIBERLITE_NVP(bio);
    }
public:
    string name;
    double age;
    vector<string> bio;
};
HIBERLITE_EXPORT_CLASS(Person)
```



paulftw/**hiberlite**

https://github.com/paulftw/hiberlite

Какие уже есть ORM для SQLite на C++?

```
TabFoo foo;
Db db(/* some arguments*/);

for (const auto& row : db(select(all_of(foo)).from(foo).where(foo.hasFun or foo.name == "joker")))
{
    int64_t id = row.id;
}
```

rbock/**sqlpp11**

https://github.com/rbock/sqlpp11

Что хочется иметь при работе с ORM?

1) не тратить кучу времени на создание БД (в коде либо во внешних редакторах)

Что хочется иметь при работе с ORM?

2) иметь встроенный язык (DSL) с отсутствием кодогенерации, макросов и прочей «черной магии»

Что хочется иметь при работе с ORM?

3) single responsibility principle - классы модели не должны иметь ORM в качестве зависимости

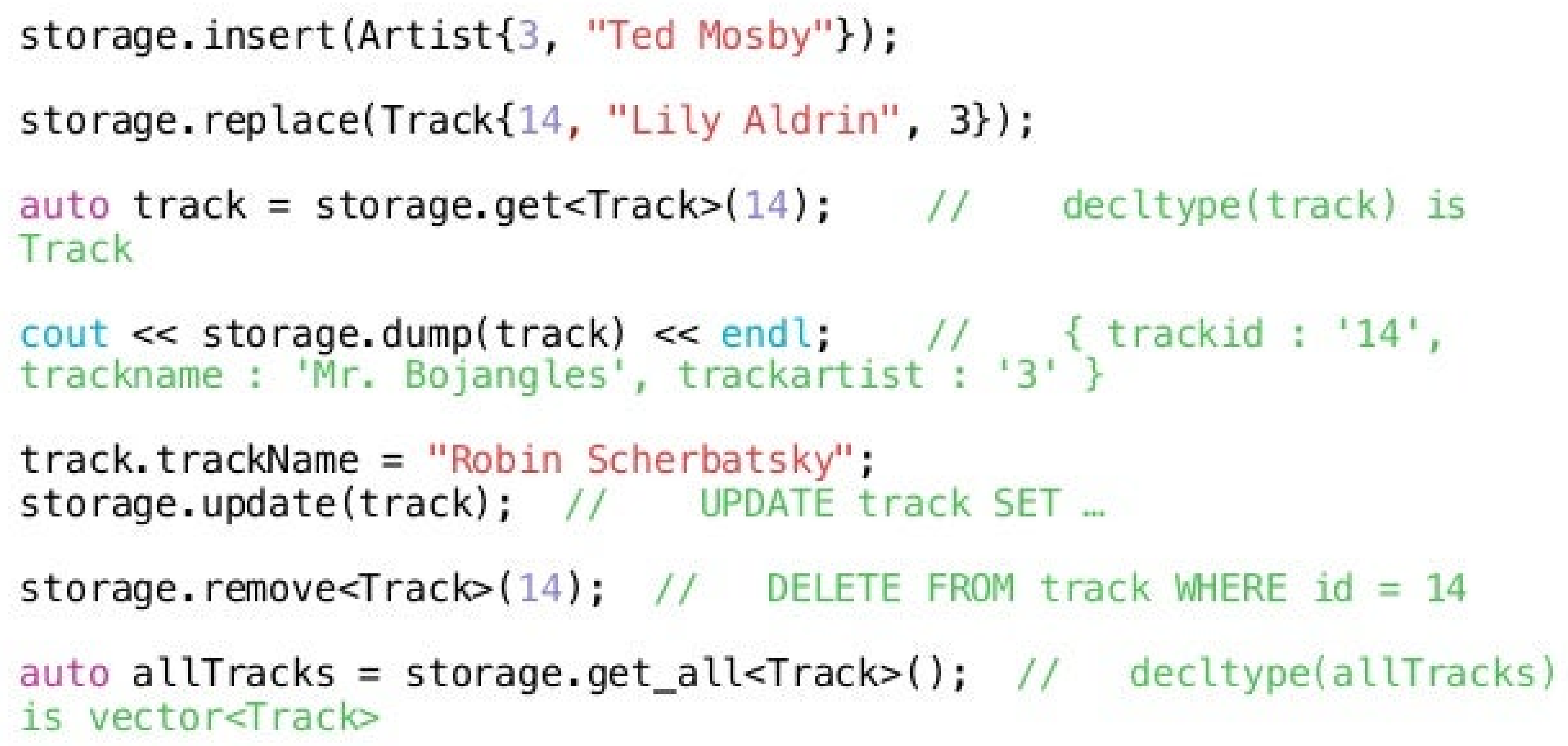
Описание схемы

```
struct User {
    int id = 0;
    std::string name;
    int tag = 0;
};

auto storage = make_storage("db.sqlite",
                   make_table("users",
                              make_column("identifier", &User::id, primary_key()),
                              make_column("name", &User::name),
                              make_column("tag", &User::tag)));


storage.sync_schema();


storage.sync_schema(true);
```

Описание схемы

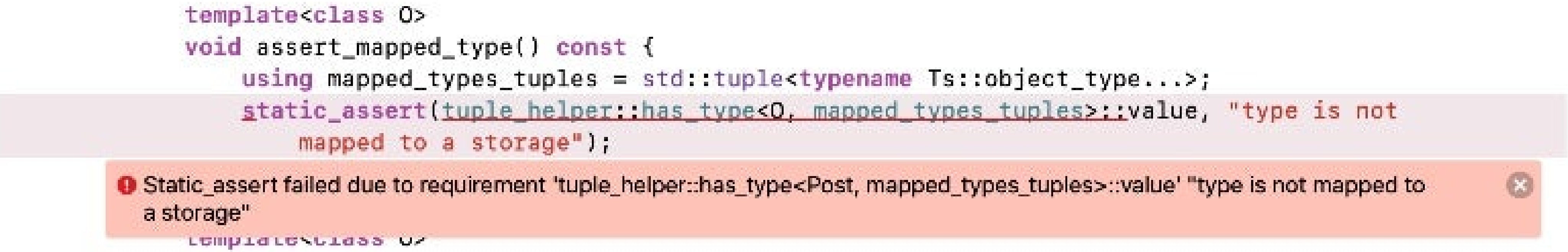
```
CREATE TABLE IF NOT EXISTS 'users' (
        ‘identifier' INTEGER PRIMARY KEY NOT NULL ,
        'name' TEXT NOT NULL ,
        'tag' INTEGER NOT NULL );
```

Банальный CRUD



```
storage.insert(Artist{3, "Ted Mosby"});

storage.replace(Track{14, "Lily Aldrin", 3});

auto track = storage.get<Track>(14);    //    decltype(track) is
Track

cout << storage.dump(track) << endl;    //    { trackid : '14',
trackname : 'Mr. Bojangles', trackartist : '3' }

track.trackName = "Robin Scherbatsky";
storage.update(track);  //    UPDATE track SET …

storage.remove<Track>(14);  //   DELETE FROM track WHERE id = 14

auto allTracks = storage.get_all<Track>();  //   decltype(allTracks)
is vector<Track>
```

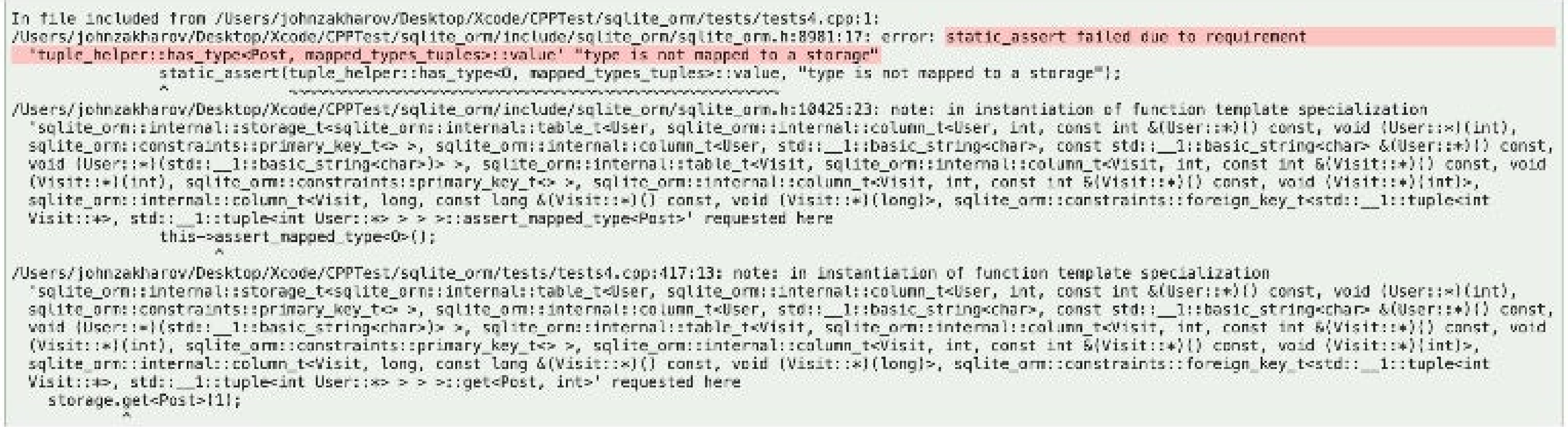
Банальный CRUD



```
template<class O>
void assert_mapped_type() const {
    using mapped_types_tuples = std::tuple<typename Ts::object_type...>;
    static_assert(tuple_helper::has_type<O, mapped_types_tuples>::value, "type is not
        mapped to a storage");
```

Static_assert failed due to requirement 'tuple_helper::has_type<Post, mapped_types_tuples>::value' "type is not mapped to a storage"

```
template<class O>
```



```
In file included from /Users/johnzakharov/Desktop/Xcode/CPPTest/sqlite_orm/tests/tests4.cpp:1:
/Users/johnzakharov/Desktop/Xcode/CPPTest/sqlite_orm/include/sqlite_orm/sqlite_orm.h:8981:17: error: static_assert failed due to requirement
  'tuple_helper::has_type<Post, mapped_types_tuples>::value' "type is not mapped to a storage"
                static_assert(tuple_helper::has_type<O, mapped_types_tuples>::value, "type is not mapped to a storage");
                ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/Users/johnzakharov/Desktop/Xcode/CPPTest/sqlite_orm/include/sqlite_orm/sqlite_orm.h:10425:23: note: in instantiation of function template specialization
  'sqlite_orm::internal::storage_t<sqlite_orm::internal::table_t<User, sqlite_orm::internal::column_t<User, int, const int &(User::*)() const, void (User::*)(int),
  sqlite_orm::constraints::primary_key_t<> >, sqlite_orm::internal::column_t<User, std::__1::basic_string<char>, const std::__1::basic_string<char> &(User::*)() const,
  void (User::*)(std::__1::basic_string<char>)> >, sqlite_orm::internal::table_t<Visit, sqlite_orm::internal::column_t<Visit, int, const int &(Visit::*)() const, void
  (Visit::*)(int), sqlite_orm::constraints::primary_key_t<> >, sqlite_orm::internal::column_t<Visit, int, const int &(Visit::*)() const, void (Visit::*)(int)>,
  sqlite_orm::internal::column_t<Visit, long, const long &(Visit::*)() const, void (Visit::*)(long)>, sqlite_orm::constraints::foreign_key_t<std::__1::tuple<int
  Visit::*>, std::__1::tuple<int User::*> > > >::assert_mapped_type<Post>' requested here
                this->assert_mapped_type<O>();
                      ^
/Users/johnzakharov/Desktop/Xcode/CPPTest/sqlite_orm/tests/tests4.cpp:417:13: note: in instantiation of function template specialization
  'sqlite_orm::internal::storage_t<sqlite_orm::internal::table_t<User, sqlite_orm::internal::column_t<User, int, const int &(User::*)() const, void (User::*)(int),
  sqlite_orm::constraints::primary_key_t<> >, sqlite_orm::internal::column_t<User, std::__1::basic_string<char>, const std::__1::basic_string<char> &(User::*)() const,
  void (User::*)(std::__1::basic_string<char>)> >, sqlite_orm::internal::table_t<Visit, sqlite_orm::internal::column_t<Visit, int, const int &(Visit::*)() const, void
  (Visit::*)(int), sqlite_orm::constraints::primary_key_t<> >, sqlite_orm::internal::column_t<Visit, int, const int &(Visit::*)() const, void (Visit::*)(int)>,
  sqlite_orm::internal::column_t<Visit, long, const long &(Visit::*)() const, void (Visit::*)(long)>, sqlite_orm::constraints::foreign_key_t<std::__1::tuple<int
  Visit::*>, std::__1::tuple<int User::*> > > >::get<Post, int>' requested here
    storage.get<Post>(1);
            ^
```

Условия

```
auto users = storage.get_all<User>(where(lesser_than(&User::id, 5)));
//  decltype(users) is vector<User>
```

Условия

```
SELECT id, first_name, last_name
FROM users
WHERE id < 5
```

Условия

```
template<class L, class R>
conditions::lesser_than_t<L, R> lesser_than(L l, R r) {
    return {std::move(l), std::move(r)};
}


template class L, class R>
struct lesser_than_t : binary_condition<L, R>, lesser_than_string {
    using self = lesser_than_t<L, R>;

    using binary_condition<L, R>::binary_condition;

    negated_condition_t<self> operator!() const {
        return {*this};
    }
};
```

Условия

```
template class L, class R>
struct binary_condition : public condition_t {
    using left_type = L;
    using right_type = R;

    left_type l;
    right_type r;

    binary_condition() = default

    binary_condition(left_type l_, right_type r_) : l(std::move(l_)), r(std::move(r_)) {}
};
```

Условия

```
auto users = storage.get_all<User>(where(lt(&User::id, 5)));
//  decltype(users) is vector<User>

auto users = storage.get_all<User>(where(c(&User::id) < 5));
```

Условия

```
template<class T>
internal::expression_t<T> c(T t) {
    return {std::move(t)};
}


template<class T, class R>
conditions::lesser_than_t<T, R> operator<(internal::expression_t<T> expr, R r) {
    return {std::move(expr.t), std::move(r)};
}


template<class L, class T>
conditions::lesser_than_t<L, T> operator<(L l, internal::expression_t<T> expr) {
    return {std::move(l), std::move(expr.t)};
}
```

Условия

```
template<class T>
struct expression_t {
    T t;

    expression_t(T t_) : t(std::move(t_)) {}

    template<class R>
    assign_t<T, R> operator=(R r) const {
        return {this->t, std::move(r)};


    assign_t<T, std::nullptr_t> operator=(std::nullptr_t) const {
        return {this->t, nullptr};
    }
};
```

Условия

```
auto users = storage.get_all<User>(where(c(&User::id) < 5 and
c(&User::firstName) == «Barney»));
```

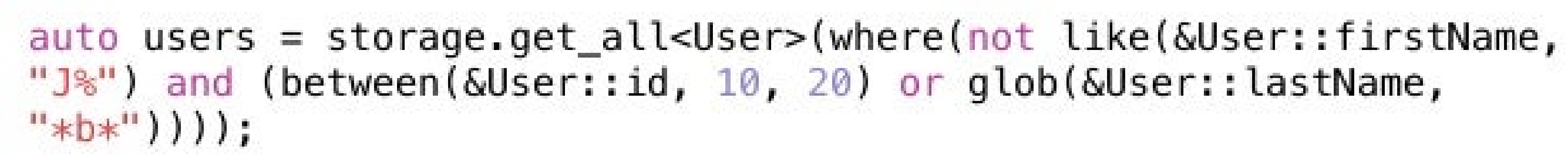
Условия

```
SELECT id, first_name, last_name
FROM users
WHERE id < 5 AND first_name = ‘Barney’
```

Условия

```
template<class L,
         class R,
         typename = typename
std::enable_if<std::is_base_of<conditions::condition_t, L>::value ||
std::is_base_of<conditions::condition_t, R>::value>::type>
and_condition_t<L, R> operator&&(L l, R r)
{
    return {std::move(l), std::move(r)};
}
```

Core functions & operators



```
auto users = storage.get_all<User>(where(not like(&User::firstName,
"J%") and (between(&User::id, 10, 20) or glob(&User::lastName,
"*b*"))));
```

Core functions & operators

```
SELECT id, first_name, last_name
FROM users
WHERE NOT (first_name LIKE ‘J%’) AND (id BETWEEN 10 AND 20 OR
last_name GLOB ‘‘*b*’)
```

Core functions & operators

```cpp
struct between_string {
    operator std::string() const {
        return "BETWEEN";
    }
};


template<class A, class T>
struct between_t : condition_t, between_string {
    using expression_type = A;
    using lower_type = T;
    using upper_type = T;

    expression_type expr;
    lower_type b1;
    upper_type b2;

    between_t(expression_type expr_, lower_type b1_, upper_type b2_) :
        expr(std::move(expr_)), b1(std::move(b1_)), b2(std::move(b2_)) {}
};


template<class A, class T>
between_t<A, T> between(A expr, T b1, T b2) {
    return {std::move(expr), std::move(b1), std::move(b2)};
}
```

Raw select

```
auto allIds = storage.select(&User::id);
//  decltype(allIds) is vector<decltype(User::id)> AKA
vector<int>
```

Raw select

```
SELECT id
FROM users
```

Raw select

```
auto rows = storage.select(columns(&User::firstName, &User::lastName),
                                   where(c(&User::id) > 250),
                                   order_by(&User::id));
//  decltype(partialSelect) is vector<tuple<string, string>>
```

Raw select

```
SELECT first_name, last_name
FROM users
WHERE id > 250
ORDER BY id
```

Raw select

```
auto rows = storage.select(columns(&Doctor::id, &Doctor::name,
&Visit::patientName, &Visit::vdate),
                           left_join<Visit>(on(c(&Doctor::id) ==
&Visit::doctorId)));
```

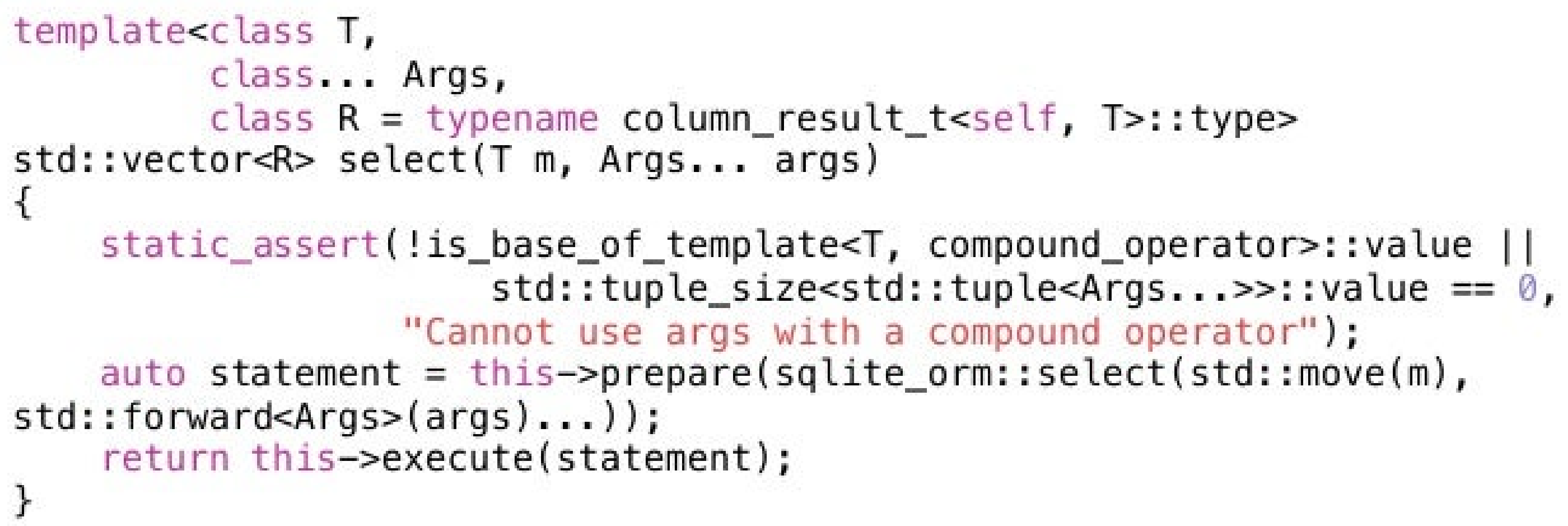
Raw select

```
SELECT doctors.doctor_id, doctors.doctor_name,
           visits.patient_name, visits.vdate
FROM doctors
LEFT JOIN visits
   ON doctors.doctor_id = visits.doctor_id;
```

Raw select

```
template<class... Args>
internal::columns_t<Args...> columns(Args... args) {
    return {std::make_tuple<Args...>(std::forward<Args>(args)...)};
}


template<class... Args>
struct columns_t {
    using columns_type = std::tuple<Args...>;

    columns_type columns;
    bool distinct = false

    static constexpr const size_t count = std::tuple_size<columns_type>::value;
};
```

Raw select



```
template<class T,
         class... Args,
         class R = typename column_result_t<self, T>::type>
std::vector<R> select(T m, Args... args)
{
    static_assert(!is_base_of_template<T, compound_operator>::value ||
                      std::tuple_size<std::tuple<Args...>>::value == 0,
                  "Cannot use args with a compound operator");
    auto statement = this->prepare(sqlite_orm::select(std::move(m),
std::forward<Args>(args)...));
    return this->execute(statement);
}
```

Raw select

```
template<class St, class T, class SFINAE = void>
struct column_result_t;

template<class St, class O, class F>
struct column_result_t<St,
                   F O::*,
                   typename std::enable_if<std::is_member_pointer<F O::*>::value
                   &&!std::is_member_function_pointer<F O::*>::value>::type>
{
    using type = F;
};

template<class St, class... Args>
struct column_result_t<St, columns_t<Args...>, void>
{
    using type = std::tuple<typename column_result_t<St, typename
std::decay<Args>::type>::type...>;
};
```

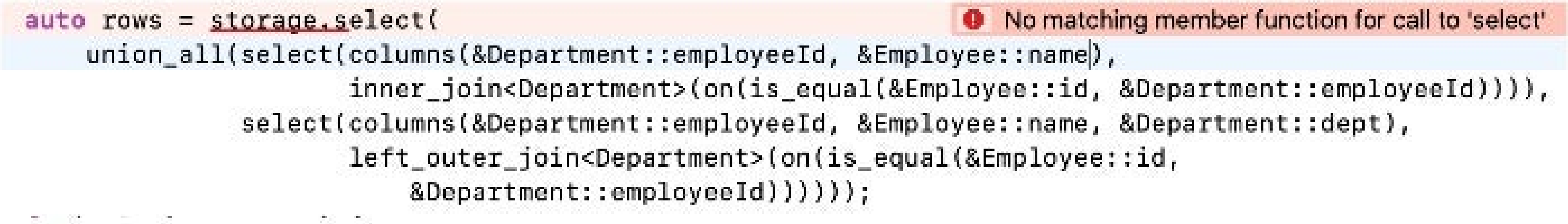
Raw select

```
auto rows = storage.select(
            union_all(select(columns(&Department::employeeId,
                                     &Employee::name,
                                     &Department::dept),
                      inner_join<Department>(on(c(&Employee::id) == &Department::employeeId))),
                      select(columns(&Department::employeeId,
                                     &Employee::name,
                                     &Department::dept),
                      left_outer_join<Department>(on(c(&Employee::id) == &Department::employeeId)))));
```

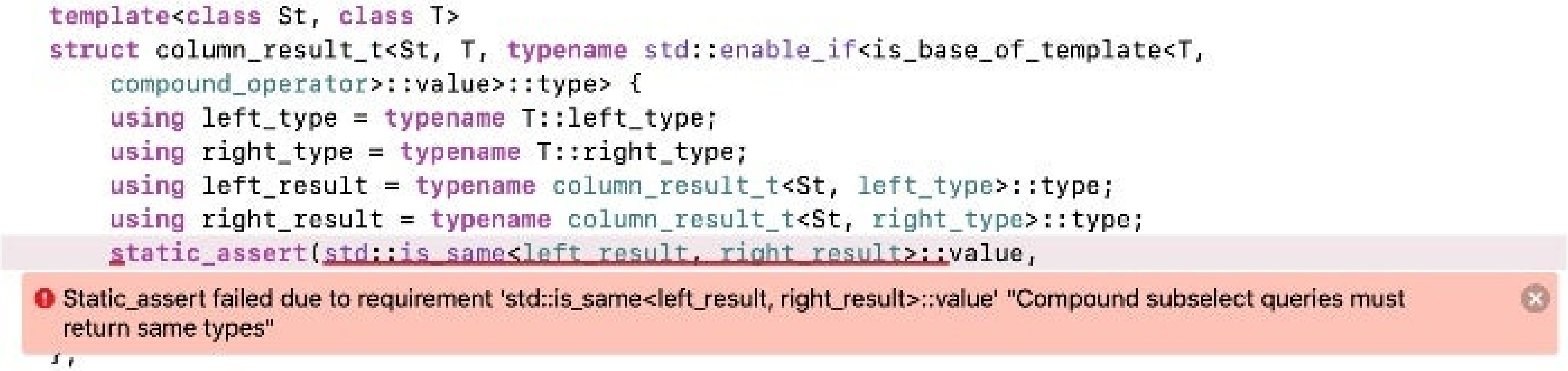
Raw select

```sql
SELECT emp_id, name, dept
FROM company
INNER JOIN department
ON company.id = department.emp_id
UNION ALL
SELECT emp_id, name, dept
FROM company
LEFT OUTER JOIN department
ON company.id = department.emp_id
```

Raw select



```
auto rows = storage.select(                                   No matching member function for call to 'select'
    union_all(select(columns(&Department::employeeId, &Employee::name),
                     inner_join<Department>(on(is_equal(&Employee::id, &Department::employeeId)))),
              select(columns(&Department::employeeId, &Employee::name, &Department::dept),
                     left_outer_join<Department>(on(is_equal(&Employee::id,
                         &Department::employeeId))))));
```



```
template<class St, class T>
struct column_result_t<St, T, typename std::enable_if<is_base_of_template<T,
    compound_operator>::value>::type> {
    using left_type = typename T::left_type;
    using right_type = typename T::right_type;
    using left_result = typename column_result_t<St, left_type>::type;
    using right_result = typename column_result_t<St, right_type>::type;
    static_assert(std::is_same<left_result, right_result>::value,
```

Static_assert failed due to requirement 'std::is_same<left_result, right_result>::value' "Compound subselect queries must return same types"

```
};
```

Prepared statements

```
auto statement = storage.prepare(select(columns(5.0,
                                                &User::id,
                                                count(&User::name)),
                                 where(c(&User::id) < 10)));

auto rows = storage.execute(statement);
```

Prepared statements

```
SELECT 5.0, id, COUNT(name)
FROM users
WHERE id < 10
```

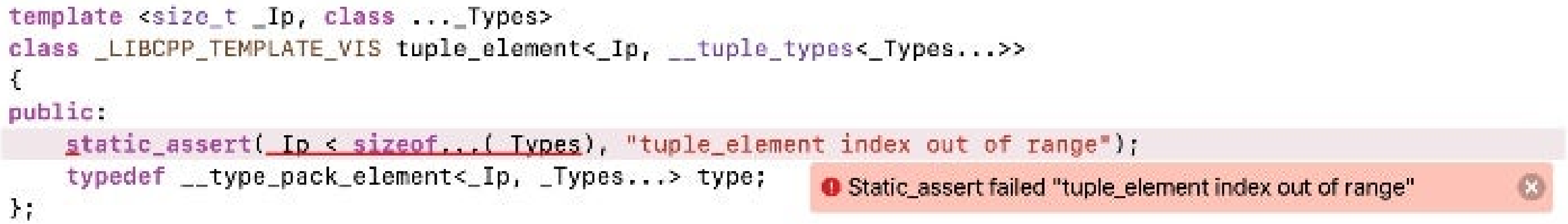
Prepared statements

```
get<0>(statement) = 4;
get<1>(statement) = 2;
```

Prepared statements

```
SELECT 4.0, id, COUNT(name)
FROM users
WHERE id < 2
```

Prepared statements



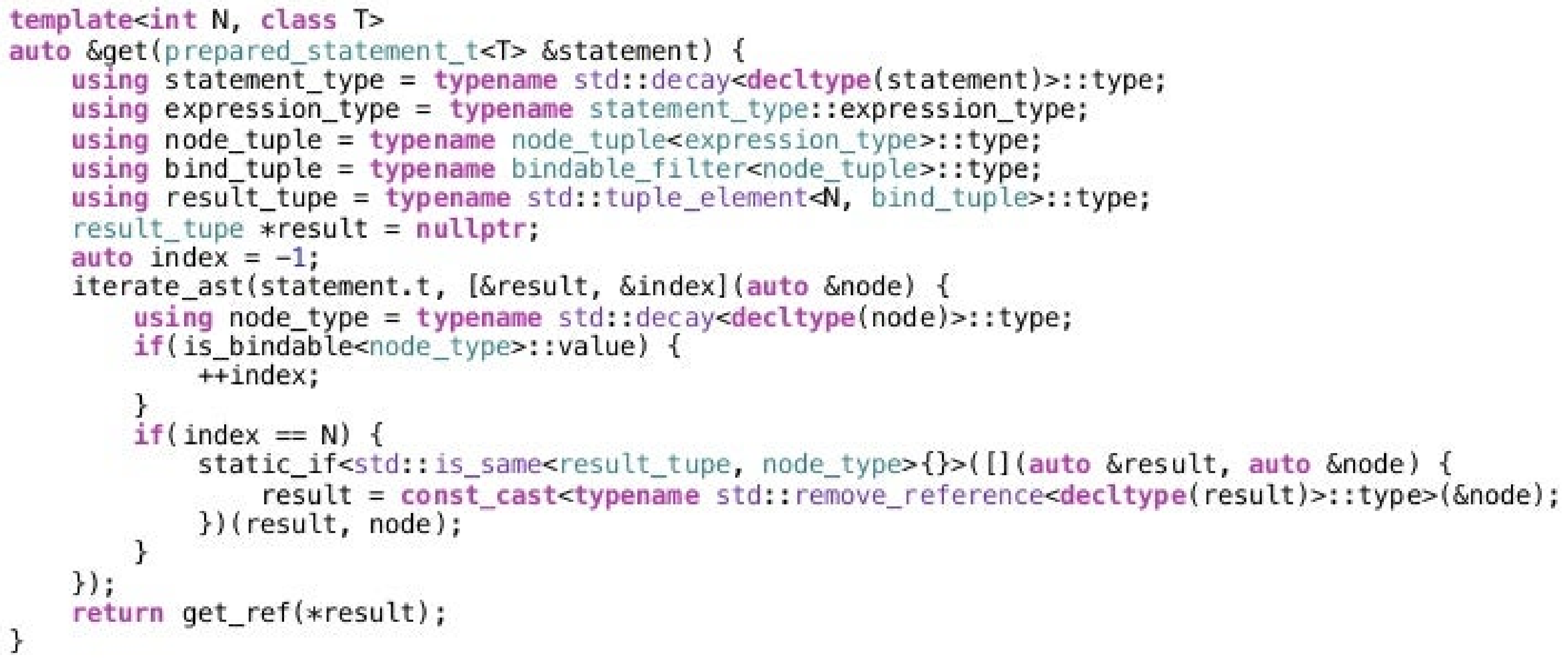
```
template <size_t _Ip, class ..._Types>
class _LIBCPP_TEMPLATE_VIS tuple_element<_Ip, __tuple_types<_Types...>>
{
public:
    static_assert(_Ip < sizeof...(_Types), "tuple_element index out of range");
    typedef __type_pack_element<_Ip, _Types...> type;
};
```

Static_assert failed "tuple_element index out of range"

Prepared statements

```
template<class T, class... Args>
struct select_t {
    using return_type = T;
    using conditions_type = std::tuple<Args...>;

    return_type col;
    conditions_type conditions;
    bool highest_level = false;
};


template<class T, class... Args>
select_t<T, Args...> select(T t, Args... args) {
    return {std::move(t), std::make_tuple<Args...>(std::forward<Args>(args)...)};
}
```

Prepared statements



```
template<int N, class T>
auto &get(prepared_statement_t<T> &statement) {
    using statement_type = typename std::decay<decltype(statement)>::type;
    using expression_type = typename statement_type::expression_type;
    using node_tuple = typename node_tuple<expression_type>::type;
    using bind_tuple = typename bindable_filter<node_tuple>::type;
    using result_tupe = typename std::tuple_element<N, bind_tuple>::type;
    result_tupe *result = nullptr;
    auto index = -1;
    iterate_ast(statement.t, [&result, &index](auto &node) {
        using node_type = typename std::decay<decltype(node)>::type;
        if(is_bindable<node_type>::value) {
            ++index;
        }
        if(index == N) {
            static_if<std::is_same<result_tupe, node_type>{}>([](auto &result, auto &node) {
                result = const_cast<typename std::remove_reference<decltype(result)>::type>(&node);
            })(result, node);
        }
    });
    return get_ref(*result);
}
```

AST iteration

```
template<class T, class L>
void iterate_ast(const T &t, const L &l) {
    ast_iterator<T> iterator;
    iterator(t, l);
}

template<class T, class SFINAE = void>
struct ast_iterator {
    using node_type = T;

    template<class L>
    void operator()(const T &t, const L &l) const {
        l(t);
    }
};

template<class T>
struct ast_iterator<
    T,
    typename std::enable_if<is_base_of_template<T,
conditions::binary_condition>::value>::type> {
    using node_type = T;

    template<class L>
    void operator()(const node_type &binaryCondition, const L &l) const {
        iterate_ast(binaryCondition.l, l);
        iterate_ast(binaryCondition.r, l);
    }
};
```

Ложка дегтя

Полезные ссылки

outlaw studio

fnc12/sqlite_orm

https://github.com/fnc12/sqlite_orm

Другие библиотеки

https://github.com/iwongu/sqlite3pp

https://github.com/SqliteModernCpp/sqlite_modern_cpp

https://github.com/SRombauts/SQLiteCpp

https://github.com/paulftw/hiberlite

https://github.com/rbock/sqlpp11

https://github.com/qicosmos/ormpp

https://www.codesynthesis.com/products/odb/