REFLECTION TS

Александр Ганюхин
Orion Innovation

ГЛАВА I

РЕФЛЕКСИЯ

ЧТО ТАКОЕ РЕФЛЕКСИЯ?

Рефлекси́я

формулировка, охватывающая широкий круг явлений и концепций, так или иначе относящихся к обращению разума [...] на самого себя.

JS

```
var person = {
    fname: "Default",
    lname: "Default",
    getFullName: function(){
        return this.fname + " " + this.lname;
    }
}

var john = {
    fname: "John",
    lname: "Doe"
}

john.__proto__ = person;

//Reflection : Iterate over the members of an object
for(var prop in john){
    console.log(prop + " : " + john[prop]);
}
```

JS

```
var person = {
    fname: "Default",
    lname: "Default",
    getFullName: function(){
        return this.fname + " " + this.lname;
    }
}

var john = {
    fname: "John",
    lname: "Doe"
}

john.__proto__ = person;

//Reflection : Iterate over the members of an object
for(var prop in john){
    console.log(prop + " : " + john[prop]);
}
```

```
fname : John
lname : Doe
getFullName : function(){
    return this.fname + " " + this.lname;
}
```

PYTHON

```
1 class my_class():
2 def foo(self):
3     print("my_class.foo")
4
5 instance = globals()["my_class"]()
6 instance.foo()
```

PYTHON

```
1 class my_class():
2 def foo(self):
3     print("my_class.foo")
4
5 instance = globals()["my_class"]()
6 instance.foo()
```

```
my_class.foo
```

РЕФЛЕКСИЯ C++

???

ГЛАВА II

РЕФЛЕКСИЯ В C++

~1991-2011

```
// from boost: boost/detail/is_incrementable.hpp
template<bool C, typename T = void>
struct enable_if { typedef T type; };
template<typename T>
struct enable_if<false, T> {};

namespace is_inc {
    typedef char (&yes)[1]; typedef char (&no)[2];

    struct tag {};
    struct any { template <class T> any(T const&); };
    tag operator++(any const &);

    template<typename T>
    static yes test(T const &);
    static no test(tag);

    template<typename _T> struct IsInc
    {
        static _T & type_value;
        static const bool value = sizeof(yes) == sizeof(test(++type_value));
    };
}
template<typename T>
struct IsInc : public is_inc::IsInc<T> {};

template<class Ty>
typename enable_if<IsInc<Ty>::value>::type sfinae_increment(Ty & arg);
```

2011-2020

```
1 template<class, class = void_t<>>
2 struct IsInc : false_type {};
3
4 template<class T>
5 struct IsInc<T, void_t<decltype( ++declval<T&>() )>>
6     : true_type {};
7
8 template<class Ty>
9 enable_if_t<IsInc<Ty>::value> sfinae_increment(Ty & arg);
```

C 2020

```
1 void sfinae_increment(auto & arg) requires requires { ++arg; };
```

РЕФЛЕКСИЯ В C++ СЕЙЧАС

Только закрытые вопросы

Возможна ли некоторая операция над объектом некоторого типа?

Конструируется ли тип из данных аргументов?

РЕФЛЕКСИЯ

"Закрытые вопросы"	"Открытые вопросы"
Возможна ли операция над объектом типа?	Какие операции возможны над типом?
Конструируется ли тип из данных аргументов?	Из каких аргументов конструируется тип?

ГЛАВА III

ПЕРВОЕ ЗНАКОМСТВО

ПЕРВОЕ ЗНАКОМСТВО

Matúš Chochlík
@matus_chochlik

Not completely up to the specs yet … but we have a reflection TS implementation:
compiler-explorer.com/z/TrYEYhqMK
github.com/matus-chochlik…
#cxx #cpp #cplusplus

9:25 AM · Nov 22, 2021 · Twitter Web App

ПЕРВОЕ ЗНАКОМСТВО

```
1 auto main() -> int {
2     using namespace std::experimental::reflect;
3
4     using mi = reflexpr(int);
5     std::cout << get_name_v<mi> << std::endl; // int
6
7     using ms = reflexpr(std);
8     std::cout << get_name_v<ms> << std::endl; // std
9 }
```

САМАЯ ПЕРВАЯ ИДЕЯ...

... СЕРИАЛИЗАЦИЯ!

10 МИНУТ СПУСТЯ...

```
 1 template<typename T>
 2 void print(T const & val);
 3
 4 struct My_object
 5 {
 6     int get_value_1() const { return 1000; }
 7     std::string get_value_2() const { return "FOO"; }
 8     float get_PI() const { return 3.1415f; }
 9 };
10
11 int main()  {
12     print(My_object{});
13     // value_1='1000' value_2='FOO' PI='3.1415'
14 }
```

РЕАЛИЗАЦИЯ...

```cpp
using namespace std::experimental::reflect;
template<typename TPublicMethodMeta, typename TObj>
auto do_call_impl(TObj const & o)
{
    auto const fx_ptr = get_pointer_v<TPublicMethodMeta>;
    return (o.*fx_ptr)();
}
template<typename TPublicMethodMeta, typename TObject>
void invoke_if_getter_and_print(TObject const & obj, std::ostream & out)
{
    constexpr std::string_view full_method_name { get_name_v<TPublicMethodMeta>  };
    if constexpr (full_method_name.starts_with("get_"))
    {
        out << full_method_name.substr(4) << "=\'"
            << do_call_impl<TPublicMethodMeta>(obj) << "\' ";
    }
}
template<size_t ... I, typename T>
void print_impl( T const & val, std::index_sequence<I...>)
{
    using public_members = get_public_member_functions_t<reflexpr(T)>;
    (invoke_if_getter_and_print<get_element_t<I, public_members>>(val, std::cout) ,...);
}
template<typename T>
void print(T const & val)
{
    using mo = reflexpr(T);
    using public_members = get_public_member_functions_t<mo>;

    print_impl(val, std::make_index_sequence<get_size_v<public_members>>{});
}
```

ВПЕЧАТЛЕНИЯ ЗА 10 МИНУТ

Похоже на то, что есть в C#/Python/других ЯП

Возможности кажутся невероятными

Безумно интересно, нужно изучать

Это стоит отдельного доклада!

ГЛАВА IV

REFLECTION TS

reflexpr(xxx)

где, xxx

```
::
type-id
namespace-name
id-expression
(expression)
function-call-expression
functional-type-conv-expression
```

МЕТА-ОБЪЕКТ

```
using meta_object = reflexpr( /* ... */ );
```

A meta-object type is an unnamed, incomplete namespace-scope class type.

A meta-object type allows inspection of some properties of the operand to reflexpr through type traits or type transformations on it.

В СТИЛЕ `type_traits`

	Полная форма	Краткая форма
Тип	`trait<mo>::type`	`trait_t<mo>`
Значение	`trait<mo>::value`	`trait_v<mo>`

КАК ВСЁ УСТРОЕНО

```
1  int main() {
2      using meta_object = reflexpr( /* ... */ );
3
4      // value
5      constexpr auto source_line = get_source_line_v<meta_object>;
6
7      // type
8      using type_t = get_reflected_type_t<meta_object>;
9
10     // type (meta-object)
11     using scope_meta_object = get_scope_t<meta_object>;
12
13     // type (sequence)
14     using mo_sequence = get_enumerators_t<meta_object>;
15 }
```

1-Й ШАГ

```
 1 constexpr int x = 123;
 2
 3 int main() {
 4     using meta_object = reflexpr(x);
 5
 6     static_assert( is_constexpr_v<meta_object> );
 7
 8     using type_meta_object = get_type_t<meta_object>;
 9     using type_of_x = get_reflected_type_t<type_meta_object>;
10     static_assert( is_same_v<int const, type_of_x> );
11
12     static_assert( *get_pointer_v<meta_object> == 123  );
13 }
```

ПРИВЕТ, МИР!

```
1  enum class Foo {
2      Hello,
3      CppRussia2022
4  };
5
6  int main() {
7      using meta_object = reflexpr(Foo);
8
9      // ????
10
11     // Hello, CppRussia2022!
12 }
```

ПРИВЕТ, МИР!

```
1 enum class Foo {
2     Hello,
3     CppRussia2022
4 };
5
6 int main() {
7     using meta_object = reflexpr(Foo);
8     using enumerators_mo = get_enumerators_t<meta_object>;
9 }
```

ПРИВЕТ, МИР!

```
 1 enum class Foo {
 2     Hello,
 3     CppRussia2022
 4 };
 5
 6 int main() {
 7     using meta_object = reflexpr(Foo);
 8     using enumerators_mo = get_enumerators_t<meta_object>;
 9
10     static_assert( get_size_v<enumerators_mo> == 2 );
11 }
```

ПРИВЕТ, МИР!

```
enum class Foo {
    Hello,
    CppRussia2022
};

int main() {
    using meta_object = reflexpr(Foo);
    using enumerators_mo = get_enumerators_t<meta_object>;

    cout << get_name_v<get_element_t<0, enumerators_mo>> << ", "
         << get_name_v<get_element_t<1, enumerators_mo>> << "!";
}
```

ПРИВЕТ, МИР!

```
1  enum class Foo {
2      Hello,
3      CppRussia2022
4  };
5
6  int main() {
7      using meta_object = reflexpr(Foo);
8      using enumerators_mo = get_enumerators_t<meta_object>;
9
10     cout << get_name_v<get_element_t<0, enumerators_mo>> << ", "
11          << get_name_v<get_element_t<1, enumerators_mo>> << "!";
12     // Hello, CppRussia2022!
13 }
```

ГЛАВА V

ВОЗМОЖНОСТИ

to_string(Enum)

to_string(Enum)

```cpp
enum class my_enum {
    one, two, three
};

string to_string(my_enum val) {
    switch (val) {
        case my_enum::one:   { return "one";   }
        case my_enum::two:   { return "two";   }
        case my_enum::three: { return "three"; }
    }
}

int main() {
    assert("three"sv == to_string(my_enum::three));
}
```

to_string(Enum)

```
1  DECLARE_ENUMERATION(my_enum, one, two, three);
2
3  string to_string(my_enum val);
4
5  int main() {
6      assert("three"sv == to_string(my_enum::three));
7  }
```

to_string(Enum)

```
1  enum class my_enum {
2      one, two, three
3  };
4
5  template<Enumeration T>
6  string to_string(T val);
7
8  int main() {
9      assert("three"sv == to_string(my_enum::three));
10 }
```

to_string(Enum)

```
1 template<Enumeration T>
2 string to_string(T val) {
3     using enumeration_mo = get_aliased_t<reflexpr(T)>;
4 }
```

to_string(Enum)

```
1 template<Enumeration T>
2 string to_string(T val) {
3     using enumeration_mo = get_aliased_t<reflexpr(T)>;
4     static_assert("my_enum"sv == get_name_v<enumeration_mo>);
5
6     using template_mo = reflexpr(T);
7     static_assert("T"sv == get_name_v<template_mo>);
8 }
```

to_string(Enum)

```
1 template<Enumeration T>
2 string to_string(T val) {
3     using enumeration_mo = get_aliased_t<reflexpr(T)>;
4 }
```

to_string(Enum)

```
1 template<Enumeration T>
2 string to_string(T val) {
3     using enumeration_mo = get_aliased_t<reflexpr(T)>;
4     using enumerators_os = get_enumerators_t<enumeration_mo>;
5 }
```

to_string(Enum)

```
1 template<Enumeration T>
2 string to_string(T val) {
3     using enumeration_mo = get_aliased_t<reflexpr(T)>;
4     using enumerators_os = get_enumerators_t<enumeration_mo>;
5 
6     return [&val]<size_t ... I>( index_sequence<I...> ) {
7 
8     }( make_index_sequence<get_size_v<enumerators_os>>{} );
9 }
```

to_string(Enum)

```
template<Enumeration T>
string to_string(T val) {
    using enumeration_mo = get_aliased_t<reflexpr(T)>;
    using enumerators_os = get_enumerators_t<enumeration_mo>;

    return [&val]<size_t ... I>( index_sequence<I...> ) {
        (cout << ... << get_name_v<get_element_t<I, enumerators_os>>);
        // onetwothree
    }( make_index_sequence<get_size_v<enumerators_os>>{} );
}
```

to_string(Enum)

```
template<Enumeration T>
string to_string(T val) {
    using enumeration_mo = get_aliased_t<reflexpr(T)>;
    using enumerators_os = get_enumerators_t<enumeration_mo>;

    return [&val]<size_t ... I>( index_sequence<I...> ) {
        string ret {};
        ignore = ( ... ||
            (
                get_constant_v<get_element_t<I, enumerators_os>> == val
                ? (ret = get_name_v<get_element_t<I, enumerators_os>>, tru
                : false
            )
        ) || (throw std::out_of_range { "No enum" }, true) ;
        return ret;
    }( make_index_sequence<get_size_v<enumerators_os>>{} );
}
```

to_string(Enum)

```
template<Enumeration T>
string to_string(T val) {
    using enumeration_mo = get_aliased_t<reflexpr(T)>;
    using enumerators_os = get_enumerators_t<enumeration_mo>;

    return [&val]<size_t ... I>( index_sequence<I...> ) {
        // ...
    }( make_index_sequence<get_size_v<enumerators_os>>{} );
}

enum class my_enum { one, two, three };

int main() {
    assert("one"sv == to_string(my_enum::one));
    assert("two"sv == to_string(my_enum::two));
    assert("three"sv == to_string(my_enum::three));
}
```

to_string(Enum)

```
template<Enumeration T>
string to_string(T val) {
    using mo = reflexpr(T);
    using enumerators = get_enumerators_t<mo>;

    constexpr auto all_enumerators = []<size_t ... I>( index_sequence<I...
        return array { get_name_v<get_element_t<I, enumerators>>... };
    }( make_index_sequence<get_size_v<enumerators>>{} );

    return all_enumerators.at( static_cast<underlying_type_t<T>>(val) );
}

enum class my_enum { one, two, three };

int main() {
    assert("one"sv == to_string(my_enum::one));
    assert("two"sv == to_string(my_enum::two));
    assert("three"sv == to_string(my_enum::three));
}
```

to_string(Enum)

```
template<Enumeration T>
string to_string_common(T val); // using fold expressions

template<Enumeration T>
string to_string_optimized(T val); // using std::array

template<Enumeration T>
string to_string(T val) {
    if constexpr ( can_use_optimized<T>() ) {
        return to_string_optimized(val);
    } else {
        return to_string_common(val);
    }
}
```

ГЛАВА V

РЕФЛЕКСИЯ ДЛЯ КЛАССОВ

СЕРИАЛИЗАЦИЯ

СЕРИАЛИЗАЦИЯ

```
 1 template<typename T>
 2 void print(T val);
 3
 4 struct My_object
 5 {
 6     int get_value_1() const { return 1000; }
 7     std::string get_value_2() const { return "FOO"; }
 8     float get_PI() const { return 3.1415f; }
 9 };
10
11 int main()  {
12     print(My_object{});
13     // value_1='1000' value_2='FOO' PI='3.1415'
14 }
```

СЕРИАЛИЗАЦИЯ

```
1 template<typename T>
2 void print(T val) {
3     using type_mo = get_aliased_t<reflexpr(T)>;
4     using public_methods_os = get_public_member_functions_t<type_mo>;
5 }
```

СЕРИАЛИЗАЦИЯ

```
1 template<typename T>
2 void print(T val) {
3     using type_mo = get_aliased_t<reflexpr(T)>;
4     using public_methods_os = get_public_member_functions_t<type_mo>;
5 
6     [&val]<size_t ... I>( index_sequence<I...> ) {
7         // ???
8     }( make_index_sequence<get_size_v<public_methods_os>>{} );
9 }
```

СЕРИАЛИЗАЦИЯ

```
1  template<typename method_mo, typename T>
2  void process_method(T val) {
3
4  }
5
6  template<typename T>
7  void print(T val) {
8      using type_mo = get_aliased_t<reflexpr(T)>;
9      using public_methods_os = get_public_member_functions_t<type_mo>;
10
11     [&val]<size_t ... I>( index_sequence<I...> ) {
12         ( process_method<get_element_t<I, public_methods_os>>(val), ... );
13     }( make_index_sequence<get_size_v<public_methods_os>>{} );
14 }
```

СЕРИАЛИЗАЦИЯ

```
1  template<typename method_mo, typename T>
2  void process_method(T val) {
3      constexpr string_view full_method_name = get_name_v<method_mo>;
4      constexpr auto        method_ptr       = get_pointer_v<method_mo>;
5
6      if constexpr ( full_method_name.starts_with("get_") ) {
7          cout << full_method_name.substr(4) << "=\'"
8               << (val.*method_ptr)() << "\' ";
9      }
10 }
11 template<typename T>
12 void print(T val) {
13     using type_mo = get_aliased_t<reflexpr(T)>;
14     using public_methods_os = get_public_member_functions_t<type_mo>;
15
16     [&val]<size_t ... I>( index_sequence<I...> ) {
17         ( process_method<get_element_t<I, public_methods_os>>(val), ... );
18     }( make_index_sequence<get_size_v<public_methods_os>>{} );
19 }
```

ОБЫЧНАЯ СЕРИАЛИЗАЦИЯ

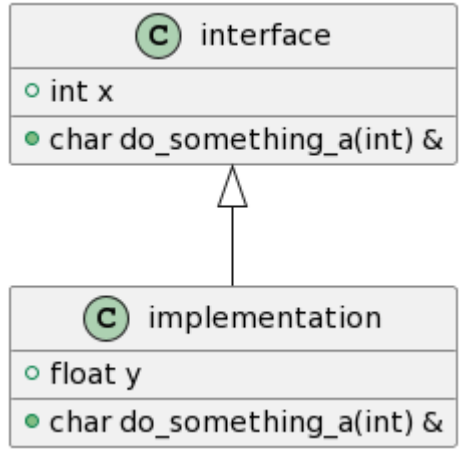
Это скучно

НЕОБЫЧНАЯ СЕРИАЛИЗАЦИЯ

SELF UML

SELF UML

```
1  struct interface {
2      virtual void do_something_a (int) & = 0;
3      int x;
4  };
5  struct implementation : interface {
6      void do_something_a (int) & override;
7      float y;
8  };
9
10 int main()  {
11     generate_uml<implementation>(); // ->
12 }
```



SELF UML

```
1 template<typename T>
2 void generate_uml() {
3     using class_mo = get_aliased_t<reflexpr(T)>;
4 }
```

SELF UML

```
template<typename T>
void generate_uml() {
    using class_mo = get_aliased_t<reflexpr(T)>;

    using bc_os = get_base_classes_t<class_mo>;
    []<size_t ... I>( index_sequence<I...> ) {
        (..., generate_uml<get_reflected_type_t<get_class_t<get_element_t<I, bc_os>>>>());
    }(make_index_sequence<get_size_v<bc_os>>{});
}
```

SELF UML

```
template<typename T>
void generate_uml() {
    using class_mo = get_aliased_t<reflexpr(T)>;

    using bc_os = get_base_classes_t<class_mo>;
    []<size_t ... I>( index_sequence<I...> ) {
        (..., generate_uml<get_reflected_type_t<get_class_t<get_element_t<I, bc_os>>>>());
        (..., (
            cout << get_name_v<get_class_t<get_element_t<I, base_classes_os>>>
                 << " <|-- " << get_name_v<class_mo> << "\n"
        ));
    }(make_index_sequence<get_size_v<bc_os>>{});
}
```

SELF UML

```
1  template<typename T>
2  void generate_uml() {
3      using class_mo = get_aliased_t<reflexpr(T)>;
4
5      using bc_os = get_base_classes_t<class_mo>;
6      /* generate_uml for bc_os */
7
8      cout << "class " << get_name_v<class_mo> << " {\n";
9
10     /* ... */
11
12     cout << "}\n";
13 }
```

SELF UML

```
template<typename T>
void generate_uml() {
    using class_mo = get_aliased_t<reflexpr(T)>;

    using bc_os = get_base_classes_t<class_mo>;
    /* generate_uml for bc_os */

    cout << "class " << get_name_v<class_mo> << " {\n";

    using public_methods_os = get_public_member_functions_t<class_mo>;
    []<size_t ... I>( std::index_sequence<I...> ) {
        ( ..., dump_method<get_element_t<I, public_methods_os>>( ) );
    }( std::make_index_sequence<get_size_v<public_methods_os>>{} );

    cout << "}\n";
}
```

SELF UML

```
1  template<typename T>
2  void generate_uml() {
3      using class_mo = get_aliased_t<reflexpr(T)>;
4
5      using bc_os = get_base_classes_t<class_mo>;
6      /* generate_uml for bc_os */
7
8      cout << "class " << get_name_v<class_mo> << " {\n";
9
10     using public_methods_os = get_public_member_functions_t<class_mo>;
11     /* dump_method for public_methods_os */
12
13     cout << "}\n";
14 }
```

SELF UML

```
template<typename T>
void generate_uml() {
    using class_mo = get_aliased_t<reflexpr(T)>;

    using bc_os = get_base_classes_t<class_mo>;
    /* generate_uml for bc_os */

    cout << "class " << get_name_v<class_mo> << " {\n";

    using public_methods_os = get_public_member_functions_t<class_mo>;
    /* dump_method for public_methods_os */

    using public_data_os = get_public_data_members_t<class_mo>;
    []<size_t ... I>( std::index_sequence<I...> ) {
        ( ..., dump_field<get_element_t<I, public_data_os>>( ) );
    }( std::make_index_sequence<get_size_v<public_data_os>>{} );

    cout << "}\n";
}
```

SELF UML

```
1  template<typename T>
2  void generate_uml() {
3      using class_mo = get_aliased_t<reflexpr(T)>;
4
5      using bc_os = get_base_classes_t<class_mo>;
6      /* generate_uml for bc_os */
7
8      cout << "class " << get_name_v<class_mo> << " {\n";
9
10     using public_methods_os = get_public_member_functions_t<class_mo>;
11     /* dump_method for public_methods_os */
12
13     using public_data_os = get_public_data_members_t<class_mo>;
14     /* dump_field for public_data_os */
15
16     cout << "}\n";
17 }
```

SELF UML

```
template<typename T>
void generate_uml() {
    using public_methods_os = get_public_member_functions_t<class_mo>;
    /* dump_method for public_methods_os */
}

template<typename mo>
void dump_method( ) {
    using parameters_os = get_parameters_t<mo>;

    cout << "  +" << get_name_v<get_type_t<mo>> << " " << get_name_v<mo> << "(";

    []<size_t ... I>( std::index_sequence<I...> ){
        ( ..., (cout << get_name_v<get_type_t<get_element_t<I, parameters_os>>>) );
    }( std::make_index_sequence<get_size_v<parameters_os>>{} );

    cout << ")";

    if ( has_lvalueref_qualifier_v<mo> ) { cout << " &"; }

    cout << "\n";
}
```

SELF UML

```
1  template<typename T>
2  void generate_uml() {
3      using class_mo = get_aliased_t<reflexpr(T)>;
4
5      using bc_os = get_base_classes_t<class_mo>;
6      /* generate_uml for bc_os */
7
8      cout << "class " << get_name_v<class_mo> << " {\n";
9
10     using public_methods_os = get_public_member_functions_t<class_mo>;
11     /* dump_method for public_methods_os */
12
13     using public_data_os = get_public_data_members_t<class_mo>;
14     /* dump_field for public_data_os */
15
16     cout << "}\n";
17 }
```

SELF UML

```
template<typename T>
void generate_uml() {
    using public_data_os = get_public_data_members_t<class_mo>;
    /* dump_field for public_data_os */
}

template<typename mo>
void dump_field( ) {
    cout << "  +" << get_name_v<get_type_t<mo>> << " " << get_name_v<mo> << "\n";
}
```

SELF UML

```
1  struct interface {
2      virtual void do_something_a (int) & = 0;
3      int x;
4  };
5  struct implementation : interface {
6      void do_something_a (int) & override;
7      float y;
8  };
9
10 int main()  {
11     generate_uml<implementation>(); // ->
12 }
```

SELF UML

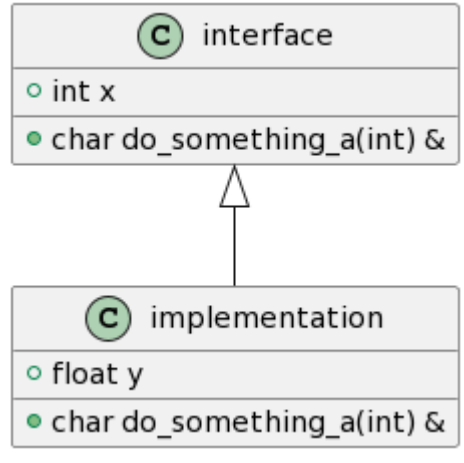
```
struct interface {
    virtual void do_something_a (int) & = 0;
    int x;
};
struct implementation : interface {
    void do_something_a (int) & override;
    float y;
};

int main()  {
    generate_uml<implementation>(); // ->
}
```

```
class interface {
    +void do_something_a(int)
    +int x
}
interface <|-- implementation
class implementation {
    +void do_something_a(int)
    +float y
}
```

SELF UML

```
class interface {
    +void do_something_a(int) &
    +int x
}
interface <|-- implementation
class implementation {
    +void do_something_a(int) &
    +float y
}
```



ФАБРИКА

ФАБРИКА

```
 1 struct A { A()      { printf("A()\n"); } };
 2 struct B { B(A)     { printf("B(A)\n"); } };
 3 struct C { C()      { printf("C()\n"); } };
 4 struct D { D(B, C) { printf("D(B, C)\n"); } };
 5
 6 template<typename T>
 7 T create();
 8
 9 int main() {
10     create<D>();
11 }
```

ФАБРИКА

```
1 template<typename T>
2 T create() {
3     using type_mo  = get_aliased_t<reflexpr(T)>;
4 }
```

ФАБРИКА

```
1 template<typename T>
2 T create() {
3     using type_mo  = get_aliased_t<reflexpr(T)>;
4
5     if constexpr (get_size_v<get_constructors_t<type_mo>> == 0) {
6         return T{};
7     }
8 }
```

ФАБРИКА

```
1  template<typename T>
2  T create() {
3      using type_mo  = get_aliased_t<reflexpr(T)>;
4
5      if constexpr (get_size_v<get_constructors_t<type_mo>> == 0) {
6          return T{};
7      } else {
8          using c_tor_mo = get_element_t<0, get_constructors_t<type_mo>>;
9          using cp_os = get_parameters_t<c_tor_mo>;
10         // ...
11     }
12 }
```

ФАБРИКА

```
template<typename T>
T create() {
    using type_mo  = get_aliased_t<reflexpr(T)>;

    if constexpr (get_size_v<get_constructors_t<type_mo>> == 0) {
        return T{};
    } else {
        using c_tor_mo = get_element_t<0, get_constructors_t<type_mo>>;
        using cp_os = get_parameters_t<c_tor_mo>;

        if constexpr (get_size_v<cp_os> == 0) {
            return T {};
        } else {
            return []<size_t ... I>( index_sequence<I...> ) {
                return T ( create<
                            get_reflected_type_t<
                             get_type_t<get_element_t<I, cp_os>>
                            >
                           >() ... );
            }( make_index_sequence<get_size_v<cp_os>>{} );
        }
    }
}
```

СОЗДАНИЕ ОБЪЕКТОВ ПО СТРОКЕ

СОЗДАНИЕ ОБЪЕКТОВ ПО СТРОКЕ

```
1 class my_class():
2     def foo(self):
3         print("my_class.foo")
4 
5 instance = globals()["my_class"]()
6 instance.foo() # my_class.foo
```

СОЗДАНИЕ ОБЪЕКТОВ ПО СТРОКЕ

```
1  struct Dummy {
2      struct Type1 { };
3      struct Type2 { };
4      struct Type3 { };
5  };
6
7  int main() {
8      auto t1 = create_from_struct_namespace<Dummy, "Type1">();
9      static_assert( is_same_v<decltype(t1), Dummy::Type1> );
10
11     auto t2 = create_from_struct_namespace<Dummy, "Type2">();
12     static_assert( is_same_v<decltype(t2), Dummy::Type2> );
13
14     auto t3 = create_from_struct_namespace<Dummy, "Type3">();
15     static_assert( is_same_v<decltype(t3), Dummy::Type3> );
16 }
```

СОЗДАНИЕ ОБЪЕКТОВ ПО СТРОКЕ

```
template<size_t N>
struct str_literal {
    constexpr str_literal(const char (&a_str)[N]) {
        copy_n(a_str, N, str);
    }
    char str[N];
};

template<typename T, str_literal name>
auto create_from_struct_namespace() {
    // ...
}

int main() {
    auto t1 = create_from_struct_namespace<Dummy, "Type1">();
    // ...
}
```

СОЗДАНИЕ ОБЪЕКТОВ ПО СТРОКЕ

```
1  template<size_t N> struct str_literal { char str[N]; };
2
3  template<str_literal name, typename type_mo, typename ... Rest>
4  auto create_value();
5
6  template<typename T, str_literal name>
7  auto create_from_struct_namespace() {
8      using struct_mo = get_aliased_t<reflexpr(T)>;
9      using public_types_os = get_public_member_types_t<struct_mo>;
10
11     return []<size_t ... I>(index_sequence<I...>) {
12         return create_value<name, get_element_t<I, public_types_os>...>();
13     }( make_index_sequence<get_size_v<public_types_os>>{});
14 }
```

СОЗДАНИЕ ОБЪЕКТОВ ПО СТРОКЕ

```
template<size_t N> struct str_literal { char str[N]; };

template<str_literal name, typename type_mo, typename ... Rest>
auto create_value() {
    if constexpr ( string_view { get_name_v<type_mo> } == string_view { name.str } ) {
        return get_reflected_type_t<type_mo> {};
    } else if constexpr ( sizeof...(Rest) > 0 ) {
        return create_value<name, Rest...>();
    } else {
        throw std::logic_error { "No such type" };
    }
}

template<typename T, str_literal name>
auto create_from_struct_namespace() {
    // ...
}
```

СОЗДАНИЕ ОБЪЕКТОВ ПО СТРОКЕ

```
1  template<size_t N> struct str_literal { char str[N]; };
2
3  template<typename T, str_literal name>
4  auto create_from_struct_namespace();
5
6  struct Dummy {
7      struct Type1 { };
8      struct Type2 { };
9      struct Type3 { };
10 };
11
12 int main() {
13     auto t1 = create_from_struct_namespace<Dummy, "Type1">();
14     static_assert( is_same_v<decltype(t1), Dummy::Type1> );
15
16     auto t1 = create_from_struct_namespace<Dummy, "Type2">();
17     static_assert( is_same_v<decltype(t1), Dummy::Type2> );
18 }
```

СОЗДАНИЕ ОБЪЕКТОВ ПО СТРОКЕ-2

СОЗДАНИЕ ОБЪЕКТОВ ПО СТРОКЕ-2

```
1  struct Dummy {
2      struct Type1 { };
3      struct Type2 { };
4      struct Type3 { };
5  };
6
7  int main() {
8      std::string s = get_object_name_from_network(); // returns "Type2"
9
10     auto dump_type = [](auto x){ cout << get_name_v<reflexpr(decltype(x))>; };
11     create_from_struct_namespace<Dummy>(s, dump_type); // Type2
12 }
```

СОЗДАНИЕ ОБЪЕКТОВ ПО СТРОКЕ-2

```
template<typename type_mo, typename ... Rest, typename Fx>
void create_and_execute(string_view name, Fx fx) {
    if ( string_view { get_name_v<type_mo> } == name ) {
        fx( get_reflected_type_t<type_mo>{} );
    } else if constexpr ( sizeof...(Rest) > 0 ) {
        create_and_execute<Rest...>(name, fx);
    } else {
        throw std::logic_error { "No such type" };
    }
};

template<typename T, typename Fx>
void create_from_struct_namespace(string_view name, Fx fx) {
    using struct_mo = get_aliased_t<reflexpr(T)>;
    using public_types_os = get_public_member_types_t<struct_mo>;

    return [&]<size_t ... I>(index_sequence<I...>) {
        return create_and_execute<get_element_t<I, public_types_os>...>(name, fx);
    }( make_index_sequence<get_size_v<public_types_os>>{});
}
```

СОЗДАНИЕ ОБЪЕКТОВ ПО СТРОКЕ-2

```
1  struct Dummy {
2      struct Type1 { };
3      struct Type2 { };
4      struct Type3 { };
5  };
6
7  int main() {
8      std::string s = get_object_name_from_network(); // returns "Type2"
9
10     auto dump_type = [](auto x){ cout << get_name_v<reflexpr(decltype(x))>; };
11     create_from_struct_namespace<Dummy>(s, dump_type); // Type2
12 }
```

ГЛАВА VI

ОГРАНИЧЕНИЯ

ОГРАНИЧЕНИЯ

Read-only рефлексия

Невозможно исследовать тела функций

Невозможно исследовать пространства имён

Невозможно исследовать аттрибуты

Невозможно исследовать шаблоны

РЕФЛЕКСИЯ ДЛЯ АТТРИБУТОВ

p1887

```
1 struct my_struct {
2     [[serialized::as("field1")]] int variable_name;
3     [[serialized::as("field2")]] int another_variable_name;
4     int this_is_not_serialized;
5 };
```

РЕФЛЕКСИЯ ДЛЯ АТТРИБУТОВ И ПРОСТРАНСТВ ИМЁН

```
1 struct my_data_type { /* ... */ };
2
3 namespace test {
4     [[gtest::test_case("my_data_type_works_good")]]
5     void my_test_case() {
6         // ...
7     }
8 }
```

CONSTEXPR REFLEXPR

p0953

```
1  struct some_type {};
2
3  int main() {
4      // reflection TS
5      using meta_object = reflexpr( some_type );
6      using public_members = get_public_member_functions_t<meta_object>;
7
8      // p0953
9      constexpr auto meta_object    = reflexpr( some_type );
10     constexpr auto public_members = meta_object.get_public_member_functior
11 }
```

ОПЕРАТОР РЕФЛЕКСИИ

p2320

```
1  struct some_type {};
2
3  int main() {
4      // reflection TS
5      using meta_object = reflexpr( some_type );
6      using type_t      = get_reflected_type_t<meta_object>;
7
8      // p2320
9      constexpr reflect::Type meta_object = ^some_type;
10     using type_t                        = :meta_object:;
11 }
```

MIRROR

```
1 int main() {
2     cout << reflect::get_name_v<reflexpr(int)> << endl;
3     cout << get_name(mirror(int)) << endl;
4 }
```

РЕФЛЕКСИЯ В C++

Рефлексия в C++ - мощно

Мы ограничены лишь воображением

Это только начало

СПАСИБО!