

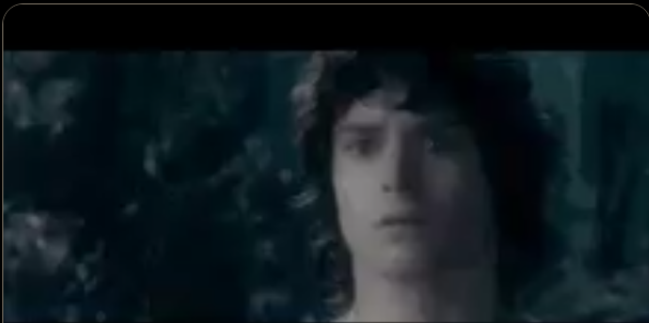
Contents

- Reflection and splicing
- Metaobjects and metadata
- Reflection APIs
- Examples
- Some use-cases



Matúš Chochlík @matus_chochlik

- "Will you look into the mirror?"
- "What will I see?"
- "...the Mirror (version 0.4.0) shows many things. Base class specifiers, constructors, destructors, data members, enumerators, ..."
[github.com/matus-chochlik...](https://github.com/matus-chochlik/mirror)



GIF

<https://github.com/matus-chochlik/mirror>

Metaobject

- A *meta-level* representation of a *base-level* entity
 - namespace, type, function, constructor, destructor, variable, constant, expression, ...
- On the *meta-level*¹ all of the above are “reified”
 - can be stored in variables, used as function arguments and return values,
 - provide the ability to write reflection algorithm libraries,
 - working at compile-time.
- We say a metaobject “reflects” the *base-level* entity

¹unlike the base-level (namespaces, constructors, etc)

Metaobject – continued

- Provides access to *metadata* describing the reflected *base-level* entity
 - type of a variable,
 - data members of a `struct`,
 - constructors or member functions of a `class`,
 - base classes of a `class`,
 - return type of a function,
 - parameters of a function,
 - enumerators in an `enum` type,
 - name of a namespace, type, function, data member, parameter, etc.
 - address of a variable, data member or member function,
 - specifiers like `virtual`, `constexpr`, `static`, `noexcept`, `public`, `protected`, `private`, etc.
 - source location²,
 - ...

²except for built-ins

Metaobject – continued

In a program a *metaobject* is a compile-time constant value of a type satisfying the following concept:

```
template <typename X>  
concept metaobject = not-really-important-here;
```

Can be used to constrain function arguments:

```
void foo(metaobject auto m) { /*...*/ }
```

or more generally in a requires clause,

```
void bar(auto m)  
    requires(metaobject<decltype(m)> &&  
             something_else(m)) {  
    /*...*/  
}
```

Reflection

- The process of obtaining metadata or metaobjects which provide metadata indirectly
- Done through a dedicated operator or language expression
- For example

```
const auto meta_int = mirror3(int);  
  
static_assert(metaobject<decltype(meta_int)>);
```

³the reflection expression

Splicing

- The reverse of reflection
- Getting back to the base-level entity reflected by a metaobject
- As in...
 - getting a type or a template,
 - getting the value of a constant,
 - getting the pointer or reference to a variable,
 - getting the pointer to a function or a member function,
 - invoking a function, constructor or operator,
 - etc.
- ...through an operation on a metaobject reflecting that *base-level* entity
- “Splicing” can also mean emitting a snippet of code involving base-level entities, reflected by metaobjects

Reflection API

- Set of (compile-time) functions operating on *metaobjects*
- Several groups
 - metaobject classification functions,
 - primitive metadata extraction,
 - metaobject sequence operations,
 - general-purpose algorithms,
 - predicates, comparators, transformation functions,
 - syntax sugar, placeholder expressions,
 - ...
- Are `constexpr` (or `constexpr`)⁴
- `noexcept` is also implied

⁴assumed, but omitted from examples in here to save space on slides

Reflection API – design goals

- The general API design stuff
- Concise and clear syntax, functional-style
- Support for as many use-cases as possible
- Provide functions handling recurring patterns and use-cases
- Support composition of function calls into bigger custom algorithms
- Proper ADL⁵; should not require excessive name qualification
- Be similar to the STL and other commonly-used libraries⁶, where it makes sense.

⁵argument-dependent lookup

⁶Boost, ...

Hello, reflection! – the obligatory first example

```
struct hello {};  
  
int main() {  
    hello world;  
  
    std::cout << get_name(mirror(hello))  
              << " , "  
              << get_name(mirror(world))  
              << " !" << std::endl;  
  
    return 0;  
}
```

"hello, world!"

Reflection API – metaobject classification functions

- Indicate *what* does a metaobject reflect
- Return bool value
- `auto reflects_object(metaobject auto m) -> bool;`
- `auto reflects_{object_sequence, named, alias, typed, scope, scope_member, enumerator, record_member, base, namespace, global_scope, type, enum, record, class, lambda, constant, variable, lambda_capture, function_parameter, callable, function, member_function, special_member_function, constructor, destructor, operator, conversion_operator, expression, parenthesized_expression, function_call_expression}(metaobject auto m) -> bool;`
- Typically used in `requires` clauses or in `if constexpr`

Reflection API – metadata retrieval functions

- Return individual “atomic” pieces of metadata
- Some are applicable to all metaobjects
 - `get_source_line`, `get_source_column`, `reflect_same`, ...
- Some only work on metaobjects reflecting specific *base-level* entities
 - where they make sense,
 - `get_name`, `get_type`, `get_scope`, `get_enumerators`, ...

Reflection API – metadata retrieval functions

- Return one of these things:
 - boolean values (`is_static`),
 - integer values (`get_source_line`),
 - other constant values⁷ (`get_constant`),
 - pointers or references⁸ (`get_pointer`),
 - strings⁹ (`get_source_file_name`, `get_name`),
 - types¹⁰ (`get_reflected_type`),
 - other metaobjects (`get_scope`, `get_data_members`).

⁷like enumerators

⁸to values, data member, functions

⁹string views actually

¹⁰more precisely `type_identity`

Reflection API – somewhere in logging...

```
auto get_display_name(metaobject auto mo) -> string_view  
    requires(reflects_named(mo));
```

```
void log_who_did_this(  
    metaobject auto mo,  
    ostream& log) {  
    if constexpr(reflects_global_scope(mo)) {  
        log << "it was the global scope!"  
    } else if constexpr(reflects_named(mo)) {  
        log << "it was " << get_display_name(mo);  
    } else {  
        log << "how the [:^#%@:]! should I know?"  
    }  
    log << " "  
        << "(" << get_source_file_name(mo)  
        << ":" << get_source_line(mo)  
        << "," << get_source_column(mo)  
        << ")";  
}
```

Metadata getters – source location

```
auto get_source_file_name(metaobject auto)  
-> string_view;
```

```
auto get_source_line(metaobject auto)  
-> unsigned long;
```

```
auto get_source_column(metaobject auto)  
-> unsigned long;
```

Metadata getters – names

```
auto get_name(metaobject auto mo) -> string_view  
    requires(reflects_named(mo));
```

"basic_string"

```
auto get_display_name(metaobject auto mo) -> string_view  
    requires(reflects_named(mo));
```

"string"

```
auto get_full_name(metaobject auto mo) -> string;
```

"std::basic_string<char, ...>"

Metadata getters – specifiers

```
auto is_constexpr(metaobject auto mo) -> bool  
    requires(reflects_variable(mo) ||  
              reflects_callable(mo));
```

```
auto is_noexcept(metaobject auto mo) -> bool  
    requires(reflects_callable(mo));
```

```
auto is_explicit(metaobject auto mo) -> bool  
    requires(reflects_constructor(mo) ||  
              reflects_conversion_operator(mo));
```

```
auto is_static(metaobject auto mo) -> bool  
    requires(reflects_variable(mo) ||  
              reflects_member_function(mo));
```

```
auto is_virtual(metaobject auto mo) -> bool  
    requires(reflects_base(mo) ||  
              reflects_destructor(mo) ||  
              reflects_member_function(mo));
```


Metadata getters – miscellaneous

```
auto is_scoped_enum(metaobject auto mo) -> bool  
    requires(reflects_type(mo));
```

```
auto uses_class_key(metaobject auto mo) -> bool  
    requires(reflects_type(mo));
```

```
auto uses_default_copy_capture(metaobject auto mo) -> bool  
    requires(reflects_lambda(mo));
```

```
auto is_explicitly_captured(metaobject auto mo) -> bool  
    requires(reflects_lambda_capture(mo));
```

```
auto has_default_argument(metaobject auto mo) -> bool  
    requires(reflects_function_parameter(mo));
```

Metadata getters – miscellaneous

```
auto has_lvalueref_qualifier(metaobject auto mo) -> bool  
    requires(reflects_member_function(mo));
```

```
auto is_implicitly_declared(metaobject auto mo) -> bool  
    requires(reflects_special_member_function(mo));
```

```
auto is_deleted(metaobject auto mo) -> bool  
    requires(reflects_callable(mo));
```

```
auto is_defaulted(metaobject auto mo) -> bool  
    requires(reflects_special_member_function(mo));
```

```
auto is_move_constructor(metaobject auto mo) -> bool  
    requires(reflects_constructor(mo));
```

Metadata getters – constants and values

```
auto get_constant(metaobject auto mo) -> const auto  
    requires(reflects_constant(mo));
```

```
auto get_value(metaobject auto mo) -> const auto&  
    requires(reflects_variable(mo));
```

```
auto get_value(metaobject auto mo, auto& obj) -> const auto&  
    requires(reflects_record_member(mo) &&  
             reflects_variable(mo));
```

Metadata getters – references and pointers

```
auto get_reference(metaobject auto mo) -> auto&  
    requires(reflects_variable(mo));
```

```
auto get_reference(metaobject auto mo, auto& obj) -> auto&  
    requires(reflects_record_member(mo) &&  
            reflects_variable(mo));
```

```
auto get_pointer(metaobject auto mo) -> auto*  
    requires(reflects_variable(mo) ||  
            reflects_function(mo));
```

```
auto get_pointer(metaobject auto mo, auto& obj) -> auto*  
    requires(reflects_record_member(mo) &&  
            reflects_variable(mo));
```

Metadata getters – invocation of callables

```
auto invoke(metaobject auto mo, auto&&... args)  
  requires(reflects_member_function(mo) &&  
           is_static(mo));
```

```
auto invoke(auto mo, auto& inst, auto&&... args)  
  requires(reflects_member_function(mo) &&  
           !is_static(mo));
```

```
auto invoke(metaobject auto mo, auto&&... args)  
  requires(reflects_constructor(mo));
```

```
auto invoke_on(auto mo, auto& inst, auto&&... args)  
  requires(reflects_member_function(mo));
```

Metadata getters – metaobjects

```
auto get_scope(metaobject auto mo)  
    requires(reflects_scoped(mo));
```

```
auto get_type(metaobject auto mo)  
    requires(reflects_typed(mo));
```

```
auto get_underlying_type(metaobject auto mo)  
    requires(reflects_enum(mo));
```

```
auto get_aliased(metaobject auto mo)  
    requires(reflects_alias(mo));
```

```
auto get_class(metaobject auto mo)  
    requires(reflects_base(mo));
```

```
auto get_subexpression(metaobject auto mo)  
    requires(reflects_parenthesized_expression(mo));
```

Metadata getters – *base-level* types

```
template <metaobject MO>  
using get_reflected_type_t = unspecified;
```

```
auto get_reflected_type(metaobject auto mo)  
-> type_identity<unspecified>  
requires(reflects_type(mo));
```

```
template <typename T>  
auto is_type(  
    metaobject auto mo,  
    type_identity<T> = {})  
-> bool requires(reflects_type(mo));
```

```
template <template <typename> class Trait>  
auto has_type_trait(metaobject auto mo)  
-> bool requires(reflects_type(mo));
```

Metaobject sequences

- Are (special kind of) metaobjects themselves
- Represent collections of other metaobjects
- Returned by many metaobject operations
 - base classes,
 - data members,
 - member functions,
 - constructors, destructors,
 - enumerators,
 - ...
- Initially the metaobject elements of a sequence are not materialized
- Only “unpacked” if necessary to improve performance

Metaobject sequences – getting sequences

```
auto get_base_classes(metaobject auto mo)  
  requires(reflects_class(mo));
```

```
auto get_captures(metaobject auto mo)  
  requires(reflects_lambda(mo));
```

```
auto get_constructors(metaobject auto mo)  
  requires(reflects_record(mo));
```

```
auto get_data_members(metaobject auto mo)  
  requires(reflects_record(mo));
```

```
auto get_destructors(metaobject auto mo)  
  requires(reflects_record(mo));
```

Metaobject sequences – getting sequences

```
auto get_enumerators(metaobject auto mo)  
    requires(reflects_enum(mo));
```

```
auto get_member_functions(metaobject auto mo)  
    requires(reflects_record(mo));
```

```
auto get_member_types(metaobject auto mo)  
    requires(reflects_record(mo));
```

```
auto get_operators(metaobject auto mo)  
    requires(reflects_record(mo));
```

```
auto get_parameters(metaobject auto mo)  
    requires(reflects_callable(mo));
```

Metaobject sequences – basic operations

Is something a sequence?

```
auto is_object_sequence(metaobject auto mo) -> bool;
```

Is there anything in the sequence?

```
auto is_empty(metaobject auto mo) -> bool  
  requires(is_object_sequence(mo));
```

How many elements are there?

```
auto get_size(metaobject auto mo) -> size_t  
  requires(is_object_sequence(mo));
```

Get the I-th element

```
template <size_t I>  
auto get_element(metaobject auto mo)  
  requires(is_object_sequence(mo));
```

Concatenate

```
auto concat(metaobject auto... mo)  
  requires((... && is_object_sequence(mo)));
```

Metaobject sequences – basic iteration

```
void for_each(metaobject auto mo, auto function)  
    requires(is_object_sequence(mo));
```

```
for_each(  
    get_enumerators(mirror(weekday)),  
    [](metaobject auto mo) {  
        cout << get_name(mo)  
            << ": "  
            << int(get_constant(mo))  
            << endl;  
    });
```

Placeholder expressions

- Placeholders
 - pre-defined constant objects – `_1`, `_2`, ...
- Placeholder expressions
 - Functions matching the metaobject operations in name taking, placeholders or other placeholder expressions as arguments
- Create objects (like lambdas) that can be called later
- Predicates, comparators, transformation functions
- Custom composite algorithms

Placeholder expressions – “huh?”

```
template <typename F>
struct placeholder_expr {
    auto operator()(auto... a) const;
    // ...
};
// CTAD guide + some specializations
```

```
constexpr const placeholder_expr<...> _1{};
constexpr const placeholder_expr<...> _2{};
```

```
template <typename X>
auto some_operation(placeholder_expr<X> e) {
    return placeholder_expr{[e](auto... a) {
        return some_operation(e(a...));
    }};
}
```

Placeholder expressions – predicates

```
reflects_named(__1);  
reflects_destructor(__1);  
is_static(__1);  
is_pure_virtual(__1);  
is_public(__1);  
is_noexcept(__1);  
is_constexpr(__1);  
is_copy_constructor(__1);  
has_rvalueref_qualifier(__1);  
uses_class_key(__1);  
is_type<int>(get_type(__1));  
has_type_trait<std::is_floating_point>(__1);
```

Placeholder expressions – comparators

```
reflect_same(__1, __2);  
get_name(__1) < get_name(__2);  
get_sizeof(__1) == get_sizeof(__2);  
get_size(get_name(__1)) > get_size(get_name(__2));
```


Placeholder expressions – transforms

```
get_type(__1);  
get_scope(__1);  
get_display_name(__1);  
get_name(get_aliased(__1));  
get_name(get_aliased(get_type(__1)));  
get_size(get_enumerators(__1));  
is_empty(get_data_members(__1));  
get_size(get_name(get_scope(get_type(__1))));  
get_type(get_element<0>(get_operators(__1)));  
get_name(get_element<1>(get_parameters(__1)));
```

Algorithms

- Implement small, specific, but non-trivial functionality
- On top of the primitive metaobject operations
- Can be easily combined in many ways into bigger, custom algorithms
- Can form and inter-operate with placeholder expressions
- Promote code re-usability
- Mostly operate on metaobject sequences
- Splicing can be done in the function objects passed to the algorithms

Algorithms – transform

Takes a sequence, returns new sequence containing metaobjects that are the result of applying a transformation function

```
auto transform(auto mo, auto function)  
  requires(is_object_sequence(mo));
```

```
auto get_parameter_types = transform(  
  get_parameters(_1),  
  get_type(_1));
```

```
auto get_base_class_types = transform(  
  get_base_classes(_1),  
  get_class(_1));
```

Algorithms – filter, remove-if

Takes a sequence, returns new sequence containing only metaobjects satisfying a predicate

```
auto filter(auto mo, auto predicate)
    requires(is_object_sequence(mo));
```

Takes a sequence, returns new sequence containing only metaobjects *not* satisfying a predicate

```
auto remove_if(auto mo, auto predicate)
    requires(is_object_sequence(mo));
```

```
auto get_virtual_functions =
    filter(get_member_functions(_1), is_virtual(_1));
```

```
auto get_nonstatic_members =
    remove_if(get_data_members(_1), is_static(_1));
```

Algorithms – count-if

Takes a sequence, returns the count of metaobjects satisfying a predicate

```
auto count_if(auto mo, auto predicate)  
    requires(is_object_sequence(mo));
```

```
auto count_public_bases =  
    count_if(  
        get_base_classes(_1),  
        is_public(_1));
```

```
auto count_integer_members =  
    count_if(  
        get_data_members(_1),  
        has_type_trait<is_integral>(get_type(_1)));
```

Algorithms – find-if, find-if-not

Takes a sequence, returns the first metaobject satisfying a predicate

```
auto find_if(auto mo, auto predicate)  
  requires(is_object_sequence(mo));
```

Takes a sequence, returns the first metaobject *not* satisfying a predicate

```
auto find_if_not(auto mo, auto predicate)  
  requires(is_object_sequence(mo));
```

```
auto find_function_foo = find_if(  
  get_member_functions(_1),  
  [](auto mo) { return has_name(mo, "foo"); });
```

```
auto find_nonstatic_member = find_if_not(  
  get_data_members(_1),  
  is_static(_1));
```

Algorithms – find-ranking

Takes a sequence, applies a query function returning a value on each metaobject. Returns the metaobject for which the value is *largest* according to a compare function¹¹.

```
auto find_ranking(  
    auto mo, auto query, auto compare)  
    requires(is_object_sequence(mo));  
  
auto find_ranking(auto mo, auto query)  
    requires(is_object_sequence(mo));
```

```
auto find_largest_data_member =  
    find_ranking(  
        get_data_members(_1),  
        get_sizeof(get_type(_1)));
```

¹¹if unspecified then less-than is used

Algorithms – get-top-value

Takes a sequence, applies a query function returning a value on each metaobject.
Returns the *value* which is *largest* according to a compare function¹².

```
auto get_top_value(  
    auto mo, auto query, auto compare)  
    requires(is_object_sequence(mo));  
  
auto get_top_value(auto mo, auto query)  
    requires(is_object_sequence(mo));
```

```
auto get_max_arity =  
    get_top_value(  
        get_member_functions(_1),  
        get_size(get_parameters(_1)));
```

¹²if unspecified then less-than is used

Algorithms – choose

Takes a sequence, finds the first metaobject for which a `condition` is satisfied. If found, returns the result of a `transform` function on that metaobject, otherwise returns the `fallback` value.

```
auto choose(  
    auto fallback,  
    auto mo  
    auto condition,  
    auto transform) -> decltype(fallback)  
requires(is_object_sequence(mo));
```

Algorithms – the gist...

- There are more such algorithms
 - fold
 - join
 - is_sorted
 - all_of, any_of, none_of
 - ...
- Named algorithms convey meaning of code better
- Typically require less typing than writing for-loops
- They may hide some compiler magic
 - filter, transform,
 - ...

Composition

- The **super-power** of named algorithms
- Together with the primitive operations and the placeholder expressions, the basic algorithms can be *combined* into bigger, custom algorithms in-place
- Some examples follow...

Are enumerators consecutive?

```
enum class digits {  
    zero = 0,  
    one,  
    two,  
    three,  
    four,  
    five,  
    six,  
    seven,  
    eight,  
    nine  
};
```

```
enum class po2s {  
    one = 1,  
    two = 2,  
    four = 4,  
    eight = 8,  
    sixteen = 16,  
    thirty_two = 32  
};
```

Are enumerators consecutive?

```
auto are_consecutive = is_sorted(  
    get_enumerators(_1),  
    [](metaobject auto l, metaobject auto r) {  
        return int(get_constant(l)) ==  
            int(get_constant(r)) - 1;  
    });
```

```
cout << are_consecutive(mirror(digits)) << endl;  
cout << are_consecutive(mirror(po2s)) << endl;
```

Output:

1
0

Find enumerator with longest name

```
void print_enum(metaobject auto mo) {  
    auto find_enum_with_longest_name =  
        find_ranking(  
            get_enumerators(_1),  
            get_size(get_name(_1)));  
  
    auto me = find_enum_with_longest_name(mo);  
  
    cout << get_name(me)  
        << ", length: "  
        << get_name(me).size()  
        << ", value: "  
        << int(get_constant(me))  
        << endl;  
}
```

Find enumerator with longest name

```
enum class weekday : int {  
    monday = 1,  
    tuesday,  
    wednesday,  
    thursday,  
    friday,  
    saturday,  
    sunday  
};
```

```
enum class month : int {  
    january = 1,  
    february,  
    march,  
    april,  
    may,  
    june,  
    july,  
    august,  
    september,  
    october,  
    november,  
    december  
};
```

```
print_enum(mirror(weekday));  
print_enum(mirror(month));
```

Output:

```
wednesday, length: 9, value: 3  
september, length: 9, value: 9
```

Does a class have overloaded functions?

```
const auto has_overloaded_functions =  
    any_of(  
        group_by(  
            get_member_functions(_1),  
            get_name(_1)),  
        [] (auto ms) {  
            return get_size(ms) > 1;   
        });
```


Does a class have overloaded functions?

```
struct foo {  
    int plus(int x) { return x; }  
  
    int plus(int x, int y) {  
        return x + y;  
    }  
    int plus(int x, int y, int z) {  
        return x + y + z;  
    }  
    int minus(int x) {  
        return -x;  
    }  
    int minus(int x, int y) {  
        return x - y;  
    }  
};
```

```
struct bar {  
    int a() {  
        return 0;  
    }  
    int b() {  
        return 1;  
    }  
    int c() {  
        return 2;  
    }  
};
```

```
cout << boolalpha << has_overloaded_functions(mirror(foo))<< endl;  
cout << boolalpha << has_overloaded_functions(mirror(bar))<< endl;
```

Output:

```
true  
false
```

Does a structure have some padding?

```
template <typename... T>
auto sum_sizeofs(type_list<T...>) -> bool {
    return (0Z + ... + sizeof(T));
}
```

```
template <typename T>
auto has_padding13() -> bool {
    return sizeof(T) >
        sum_sizeofs(extract_types14(transform(
            filter(get_data_members(mirror(T)),
                not_(is_static(_1))),
            get_type(_1))));
}
```

¹³is bigger than the sum of its parts?

¹⁴note that this operation involves splicing – getting base-level types from meta-types

Does a structure have some padding?

```
struct S1 {  
    int i;  
    float f;  
};
```

```
struct S2 {  
    char c;  
    double d;  
};
```

```
template <typename T>  
void print_has_padding() {  
    cout << get_name(remove_all_aliases(mirror(T)))  
        << ": " << (has_padding<T>()) ? "has some" : "has no"  
        << " padding\n";  
}
```

```
print_has_padding<S1>();  
print_has_padding<S2>();
```

Output:

S1: has no padding

S2: has some padding

Are data members sorted by size?

```
struct foo {  
    double d;  
    int i;  
    short s;  
    char c;  
};
```

```
struct bar {  
    int i;  
    float f;  
    long l;  
    bool b;  
};
```

```
const auto are_sorted_by_size = is_sorted(  
    get_data_members(_1),  
    get_sizeof(_1) < get_sizeof(_2));
```

```
cout << boolalpha << are_sorted_by_size(mirror(foo)) << endl;  
cout << boolalpha << are_sorted_by_size(mirror(bar)) << endl;
```

true

false

Hello again!

```
enum class greeting {  
    hello, world, of, reflection  
};
```

```
cout << join15(  
    get_enumerators16(mirror17(greeting)18),  
    to_string(get_name19(_120))21,  
    string(" ")) << "!\n";
```

¹⁵algorithm

¹⁶metaobject sequence getter

¹⁷reflection operator

¹⁸reflection in action – returns a metaobject

¹⁹metadata (name) getter

²⁰placeholder

²¹placeholder expression

Enumeration conversions – enum-to-string

```
template <typename E>
auto enum_to_string(E e) noexcept
    -> string_view {

    return choose(
        string_view{},
        get_enumerators(mirror(E)),
        has_value(_1, e),
        get_name(_1));
}
```

Enumeration conversions – string-to-enum

```
template <typename E>
auto string_to_enum(string_view s) noexcept
    -> optional<E> {

    return choose(
        optional<E>{},
        get_enumerators(mirror(E)),
        has_name(_1, s),
        get_value(_1));
}
```

Calculation of interface revision id

```
template <typename Intf>
struct versioned_interface {
public:
    static auto revision_id() noexcept -> hash_t {
        return fold(
            filter(
                get_member_functions(mirror(Intf)),
                is_virtual(_1)),
            get_hash(_1) ^ get_hash(get_type(_1)),
            [](auto... h) { return (... ^ h); });
    }
};
```


Calculation of interface revision id

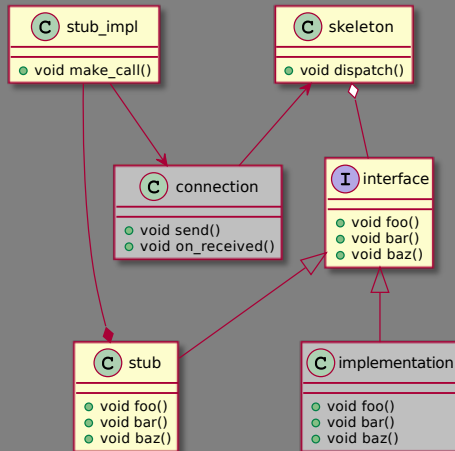
```
// initially
struct operations : versioned_interface<operations> {
    virtual void foo() = 0;
    virtual void bar(int) = 0;
    virtual auto baz(bool, bool) -> bool = 0;
};
cout << hex << operations::revision_id() << endl;
```

d423c43e4eabdc

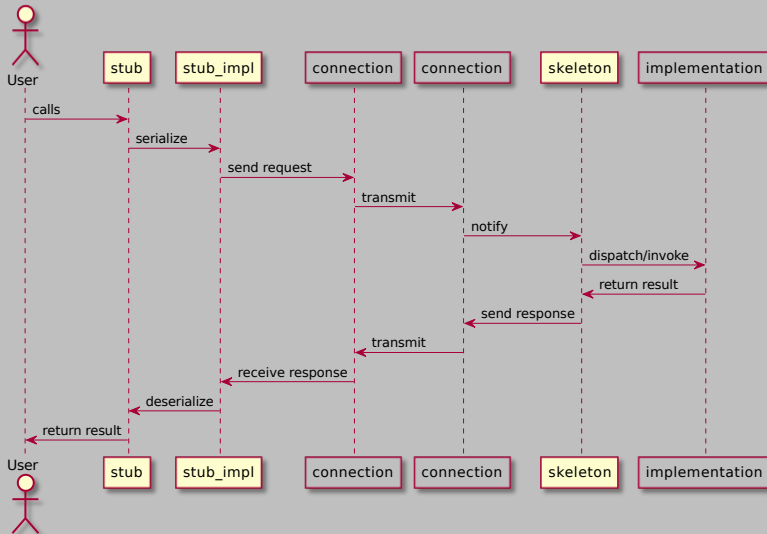
```
// later
struct operations : versioned_interface<operations> {
    virtual void foo() = 0;
    virtual void bar(long) = 0;
    virtual auto baz(bool, bool) -> int = 0;
};
cout << hex << operations::revision_id() << endl;
```

ef4203c133db7600

Remote procedure calls – class overview



Remote procedure calls – synchronous call sequence



RPC stubs/skeletons – the interface

```
struct calculator {  
    virtual float add(float, float) = 0;  
    virtual float subtract(float, float) = 0;  
    virtual float multiply(float, float) = 0;  
    virtual float divide(float, float) = 0;  
    virtual float negate(float) = 0;  
    virtual float invert(float) = 0;  
};
```

RPC stubs/skeletons – the stub

```
class calculator_stub : public calculator {  
private:  
    rpc_stub_impl _impl;  
public:  
    float add(float l, float r) final {  
        return _impl.make_call(  
            mirror((calculator::add(l, r)))22, l, r);  
    }  
  
    float subtract(float l, float r) final {  
        return _impl.make_call(  
            mirror((calculator::subtract(l, r))), l, r);  
    }  
  
    float multiply(float l, float r) final;  
    float divide(float l, float r) final;  
    float negate(float x) final;  
    float invert(float x) final;  
};
```

RPC stubs/skeletons – the generic stub implementation

```
class rpc_stub_impl {  
    template <typename T>  
    auto _deserialize(packet&, type_identity<T>) -> T;  
public:  
    auto make_call(metaobject auto mo, auto&... args)  
        requires(reflects_function_call_expression(mo)) {  
        packet request;  
        _serialize(request, mo, args...);  
        packet response{_send_and_receive(request)};  
  
        return _deserialize(  
            response,  
            get_reflected_type(  
                get_type(  
                    get_callable(  
                        get_subexpression(mo))))));  
    }  
};
```

RPC stubs/skeletons – the skeleton interface

```
struct rpc_skeleton {  
    virtual void dispatch(  
        packet& request,  
        packet& response) = 0;  
};
```

- Could be plugged-into a network connection
- Handles incoming data
- After call is finished the connection can send the response

RPC stubs/skeletons – the skeleton implementation²³

```

template <typename Intf>
class rpc_skeleton_impl : public rpc_skeleton {
private:
    std::unique_ptr<Intf> _impl;
public:
    void dispatch(packet& request, packet& response) final {
        const auto method_id{_get_method_id(request)};

        for_each(get_member_functions(mirror(Intf)),
            [&](auto mf) {
                if(_get_method_id(mf) == method_id) {
                    auto params = make_value_tuple(
                        transform(get_parameters(mf), get_type(_1)));

                    deserialize(params, request);
                    auto result = apply(mf, *_impl, params);
                    serialize(method_id, result, response);
                }
            });
    }
};

```


Scripting – is fun, hand-coding bindings, not so much

```
void add_to(chaiscript::ChaiScript& chai, auto mos)
    requires(is_object_sequence(mos));
```

```
void do_add_to(
chaiscript::ChaiScript& chai,
metaobject auto mo24,
metaobject auto ms25) {
    if constexpr(reflects_object_sequence(mo)) {
        /* ... */
    } else if constexpr(reflects_base(mo)) {
        /* ... */
    } else {
        const string name{get_name(mo)};
        if constexpr(reflects_variable(mo)) {
        } else if constexpr(reflects_constructor(mo)) {
        } else if constexpr(reflects_function(mo)) {
        } else if constexpr(reflects_record(mo)) {
        }
    }
}
```

Scripting – handling sequences²⁶

```
void do_add_to(  
    chaiscript::ChaiScript& chai,  
    metaobject auto mo,  
    metaobject auto ms) {  
    if constexpr(reflects_object_sequence(mo)) {  
        for_each(mo, [&](auto me) {  
            do_add_to(chai, me, ms);  
        });  
    } else if constexpr(reflects_base(mo)) {  
        /* ... */  
    } else {  
        /* ... */  
    }  
}
```

²⁶ “That’s easy!” – sir Robin of Camelot

Scripting – base classes

```
template <typename Base, typename Derived>
void add_base_class(
    chaiscript::ChaiScript& chai,
    type_identity<Base>,
    type_identity<Derived>) {
    chai.add(chaiscript::base_class<Base, Derived>());
}
```

```
void do_add_to(
    chaiscript::ChaiScript& chai,
    metaobject auto mo,
    metaobject auto ms) {
    if constexpr(reflects_object_sequence(mo)) {
    } else if constexpr(reflects_base(mo)) {
        add_base_class(
            chai,
            get_reflected_type(get_class(mo)),
            get_reflected_type(ms));
    } else {
        /* ... */
    }
}
```

Scripting – variables and data members

```
const string name{get_name(mo)};

if constexpr(reflects_variable(mo)) {
    if constexpr(reflects_record_member(mo)) {
        if constexpr(is_public(mo)) {
            chai.add(chaiscript::fun(
                get_pointer(mo)), name);
        }
    } else {
        chai.add(chaiscript::var(
            get_reference(mo)), name);
    }
}
```

Scripting – constructors

```
template <typename T, typename... P>
void add_constructor(
    chaiscript::ChaiScript& chai,
    type_identity<T>,
    type_list<P...>,
    const string& name) {
    chai.add(chaiscript::constructor<T(P...)>(), name);
}
```

```
const string name{get_name(mo)};

if constexpr(reflects_constructor(mo)) {
    if constexpr(is_public(mo)) {
        add_constructor(
            chai,
            get_reflected_type(get_scope(mo)),
            extract_types(transform(
                get_parameters(mo),
                get_type(_1))),
            name);
    }
}
```

Scripting – functions and operators

```
template <typename From, typename To>
void add_conversion(
    chaiscript::ChaiScript& chai,
    type_identity<From>,
    type_identity<To>) {
    chai.add(chaiscript::type_conversion<From, To>());
}
```

```
if constexpr(reflects_function(mo)) {
    if constexpr(reflects_record_member(mo) && is_public(mo)) {
        if constexpr(reflects_conversion_operator(mo)) {
            if constexpr(!is_deleted(mo)) {
                add_conversion(
                    chai,
                    get_reflected_type(get_scope(mo)),
                    get_reflected_type(get_type(mo)));
            }
        } else {
            if constexpr(!is_deleted(mo)) {
                chai.add(chaiscript::fun(get_pointer(mo)), name);
            }
        }
    } else {
        chai.add(chaiscript::fun(get_pointer(mo)), name);
    }
}
```

Scripting – types and classes

```
template <typename T>
void add_type(
    chaiscript::ChaiScript& chai,
    type_identity<T>,
    const string& name) {
    chai.add(chaiscript::user_type<T>(), name);
}
```

```
const string name{get_name(mo)};

if constexpr(reflects_record(mo)) {
    add_type(chai, get_reflected_type(mo), name);
    do_add_to(chai, get_base_classes(mo), mo);
    do_add_to(chai, get_member_types(mo), mo);
    do_add_to(chai, get_data_members(mo), mo);
    do_add_to(chai, get_constructors(mo), mo);
    do_add_to(chai, get_member_functions(mo), mo);
    do_add_to(chai, get_operators(mo), mo);
}
```

Scripting – Monty C++'s Flying Circus

```
class scene {  
public:  
    void person_says(  
        const person&, string_view line);  
    void person_relocates(  
        person& p, string_view how);  
    void event_happens(  
        string_view what);  
    void pause();  
};
```

```
class location {  
public:  
    location(string name, scene&);  
    auto name() const -> string_view;  
};
```

```
class person {  
public:  
    person(string name);  
  
    auto name() const -> string_view;  
    auto current_location() -> auto&;  
    auto enter(location&);  
    auto is_thrown_to(location&);  
    void say(const string&);  
};
```

```
class king : public person {  
public:  
    king(string name);  
};
```

```
class mysterious_force {  
public:  
    mysterious_force(location&);  
    void throw_into_chasm(person&);  
};
```


Scripting – set the scene...

```

scene at_the_bridge;
chaiscript::ChaiScript chai;

add_to(
  chai, make_sequence(
    mirror(mysterious_force),
    mirror(scene), mirror(location),
    mirror(person), mirror(king), mirror(at_the_bridge)));

```

```

chai27(R"(
  var road = location("road leading to the bridge", at_the_bridge);
  var bridge = location("bridge of death", at_the_bridge);
  var chasm = location("chasm", at_the_bridge);

  var the_force = mysterious_force(chasm);

  var bridgekeeper = person("the Bridgekeeper");
  var king_arthur = king("king Arthur");
  var sir_lancelot = person("sir Lancelot");
  var sir_robin = person("sir Robin");
  var sir_galahad = person("sir Galahad");
  var sir_bedeveere = person("sir Bedeveere");
)");

```

²⁷should have done this in Python!

Scripting – ...and *action!*

```
chai(R"(
  bridgekeeper.enter(bridge);
  king_arthur.enter(road);
  sir_lancelot.enter(road);
  sir_robin.enter(road);
  sir_galahad.enter(road);
  sir_bedevever.enter(road);

  bridgekeeper.say(
    "Stop."
    "Who would cross the Bridge of Death "
    "must answer me these questions three, "
    "ere the other side he see.");
  sir_lancelot.say(
    "Ask me the questions, bridgekeeper. "
    "I am not afraid.");
  // ...
  the_force.throw_into_chasm(sir_robin);
  sir_robin.say("Auuuuuuuugh.");

  // ...
)
```

“Smart” concept definition – featuring CTRE!

```
template <typename T>
concept very_smart_integer28 = ctre_match<
    "((signed|unsigned) )?"\
    "((long long|long|short)( int)?|int)"
>(get_name(remove_all_aliases(mirror(T))));

auto add(very_smart_integer auto l, very_smart_integer auto r) {
    return l + r;
}
```

```
cout << add(1U, 2U) << endl;
cout << add(short(3), short(4)) << endl;
cout << add(21, 21) << endl;
cout << add(400ULL, 20ULL) << endl;
```

```
cout << add(true, false) << endl; // NOPE!
```

²⁸don't try this at home

Conclusions

- Reflection is *fun*
- Reflection is *useful*
- Reflection code can be readable
- Reflection code can be straightforward to write
- Reflection APIs can provide many non-trivial, reusable tools

There are so many use-cases – details in other talks

- Implementing the *factory* pattern
- Auto-registering with script language bindings
- Generating GUIs for visualization and data input
- Serialization and deserialization
- Implementing RPC/RMI stubs and skeletons
- Generating UML diagrams from code
- Generating DB system queries
- Fetching data from databases into C++ structures
- Generating `boost::spirit` parsers and formatters
- Parsing of configuration files
- Parsing of command-line arguments
- Mapping of URL arguments to function arguments
- ...

Where does all this come from?

- The *Mirror* library
 - primitive and sequence operations,
 - algorithms,
 - placeholder expressions,
 - examples and use-cases²⁹,
 - integration with other projects³⁰,
 - ...
- On top of the reflection TS implementation in `clang`
- There is much more than what fits into this talk
- See the reference (links below)

²⁹including bigger ones

³⁰CTRE, rapidjson, chaiscript, sqlite3, etc.

Links, shameless plugs, etc.

- *This presentation* – <https://matus-chochlik.github.io/mirror/latex/cpponseas.pdf>
- *The Mirror library repository* – <https://github.com/matus-chochlik/mirror>
- *The Mirror library reference (W.I.P.)* –
<https://matus-chochlik.github.io/mirror/doxygen/>
- *Reflection TS in clang* – <https://github.com/matus-chochlik/llvm-project>
- *Reflection TS draft* – <https://cplusplus.github.io/reflection-ts/draft.pdf>

What's next for reflection?

- More splicing
- Code fragment splicing
- Splicing *identifiers* depending on metaobjects³¹
- Support for more use-cases

³¹maybe even with custom formatting

That's all folks. . .

Thanks for your attention.
Happy to answer any additional questions.

I lied – There is more



Additional API³² – Operation name enums & generic functions

```
enum class trait {
    reflects_object,
    // ...
    reflects_expression,
    // ...
    is_call_operator_const,
    is_volatile,
    // ...
    is_public,
    is_static,
    is_virtual,
    // ...
    uses_default_copy_capture,
    // ...
};
```

```
template <trait T>
auto has_trait(
    metaobject auto) -> bool;
```

```
enum class operation {
    // ...
    get_name,
    get_type,
    // ...
    get_enumerators,
    // ...
};
```

```
template <operation O>
auto is_applicable(
    metaobject auto) -> bool;
```

```
template <operation O>
auto apply(metaobject auto);
```

```
template <operation O>
auto try_apply(metaobject auto)
    -> optional<...>;
```

Additional API – Print all metaobject traits

```
void print_traits(metaobject auto mo) {  
    const auto mes = get_enumerators(mirror(trait));  
    const auto maxl =  
        get_top_value(mes, get_size(get_name(_1)));  
  
    cout << "traits of "  
        << get_display_name(mo) << "\n";  
  
    for_each(mes, [&](metaobject auto me) {  
        cout << "  " << get_name(me)  
            << ": "  
            << string(maxl - get_name(me).size(), ' ')  
            << boolalpha  
            << has_trait<get_constant(me)>(mo)  
            << "\n";  
    });  
    cout << endl;  
}
```

```
traits of std::string
reflects_object: true
reflects_object_sequence: false
reflects_named: true
reflects_alias: true
reflects_typed: false
reflects_scope: true
reflects_scope_member: true
reflects_enumerator: false
reflects_record_member: false
reflects_base: false
reflects_namespace: false
reflects_global_scope: false
reflects_type: true
reflects_enum: false
reflects_record: true
...
```

Parsing command-line arguments – into a structure

```
class program_arg {
public:
    auto next() -> program_arg;
    auto is_long_tag(...) -> bool;
    operator string_view();
};
```

```
class program_args {
public:
    program_args(int, const char**);
    auto begin();
    auto end();
};
```

```
template <typename T>
bool parse(
    T& opts,
    const program_args& args);
```

```
struct options {
    string message{"Hi world!"};
    milliseconds interval{500};
    int count{3};
};
```

```
int main(
    int argc,
    const char** argv) {
    const program_args
        args{argc, argv};
    options opts;
    if(parse(opts, args)) {
        // do something
        return 0;
    }
    return 1;
}
```

```
template <typename T>
bool parse(T& opts, const program_args& args) {
    bool parsed = true;
    for(const auto& arg : args) {
        for_each(get_data_members(mirror(T)), [&](auto mdm) {
            if(arg.is_long_tag(get_name(mdm))) {
                if(const auto opt{from_string(
                    arg.next(), get_reflected_type(get_type(mdm)))}) {
                    get_reference(mdm, opts) = opt.value();
                } else {
                    std::cerr << "invalid value '" << arg.next()
                        << "' for option " << arg
                        << "!" << std::endl;
                    parsed = false;
                }
            }
        });
    }
    return parsed;
}
```

MOAR “smart” concepts!

```
struct excellent {  
    void foo() {}  
    void bar() {}  
    void baz() {}  
};
```

```
template <typename T>  
concept has_foo_and_such = any_of(  
    get_member_functions(mirror(T)),  
    ctref_match<"foo|bar|baz">(get_name(_1)));
```

```
void foofunction(has_foo_and_such auto) {  
    cout << "this is excellent!" << endl;  
}
```

```
foofunction(excellent{});  
foofunction(std::string{}); // ERROR
```


Min/max enumerator value

```
auto min_enum = get_top_value(  
    get_enumerators(_1),  
    get_constant(_1),  
    std::greater<>{});
```

```
auto max_enum = get_top_value(  
    get_enumerators(_1),  
    get_constant(_1),  
    std::less<>{});
```

```
auto print_info = [&](auto me) {  
    cout << get_name(me)  
        << ":  min(" << enum_to_string(min_enum(me))  
        << "), max(" << enum_to_string(max_enum(me))  
        << ")" << std::endl;  
};
```

Are enumerators bitfield bits?

```
const auto is_bitfield_enum =  
    is_sorted(get_enumerators(_1), [](auto l, auto r) {  
        if constexpr(has_type_trait<is_signed>(  
            get_underlying_type(get_type(l)))) {  
            return false;  
        } else {  
            auto to_underlying = [](auto e) {  
                using U = underlying_type_t<decltype(e)>;  
                return static_cast<U>(e);  
            };  
            return to_underlying(get_constant(l)) << 1U ==  
                to_underlying(get_constant(r));  
        }  
    });
```

Have classes the same structure?

```
auto have_same_structure = [](  
    metaobject auto ml,  
    metaobject auto mr) {  
  
    const auto structure_hash = fold(  
        get_data_members(_1),  
        get_hash(get_type(_1)),  
        [](auto... h) {  
            return (... ^ h);  
        });  
  
    return structure_hash(ml) ==  
        structure_hash(mr);  
};
```

Have classes the same structure?

```
struct foo {  
    int i;  
    float f;  
    std::string s;  
};
```

```
struct bar {  
    int j;  
    float e;  
    std::string z;  
};
```

```
struct baz {  
    long l;  
    double d;  
    char c;  
    bool b;  
};
```

```
cout << boolalpha  
      << have_same_structure(mirror(foo), mirror(bar))  
      << endl;  
cout << boolalpha  
      << have_same_structure(mirror(foo), mirror(baz))  
      << endl;
```

Output:

true

false