Compile-time Reflection, Serialization and ORM

Yu Qi
qicosmos@163.com

Outline

- **Concepts of reflection**
- Implementation of compile-time reflection
- Application of compile-time reflection
- Prospect

The Essence of Reflection

- Reflection
  - a mechanism that gets internal information of a class by metadata.
  - get the type of an object, get the fields and methods by metadada.

- Metadata
  - data that provides information about other data.

meatadata describes other data. **self description**

The Essence of Reflection

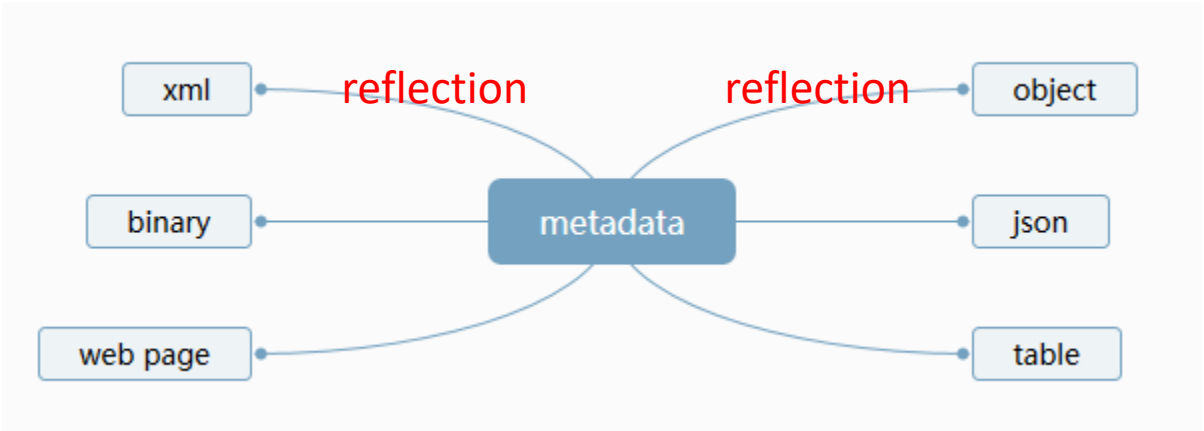
```
<xs:schema
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="country" type="Country">
     <xs:complexType name="Country">
      <xs:sequence>
       <xs:element name="name" type="xs:string"/>
       <xs:element name="population" type="xs:decimal"/>
      </xs:sequence>
     </xs:complexType>
   </xs:element>
</xs:schema>
```

	Name	Type
1	id	Integer
2	name	Text
3	age	Integer

classic metadata

name, type, sequence

The Utility of Reflection



mapping metadata to any other data format

metadata is the key point

Benefits of Reflection

- Simplification
  - cut the complexity
- Flexibility
  - change the behavior without any modification of an exist object
- Decoupling
  - decouple visit from the metadata of an object

Benefits of Reflection

If no reflection

```
void Serialize(Archive &ar)
{
    ar.Serialize("vec", vec);
    ar.Serialize("ls", ls);
    ar.Serialize("st", st);
    ar.Serialize("deq", deq);
    ar.Serialize("mp", mp);
    ar.Serialize("set", set);
    ar.Serialize("un_mp", un_mp);
    ar.Serialize("un_set", un_set);
}
```

duplicate

overelaborate

error prone

should be automatic!

Benefits of Reflection

```
struct person
{
        std::string     name;
        int64_t          age;
};

REFLECTION(person, name, age);
```

```
person p = { "tom", 20 };
iguana::string_stream ss;
iguana::json::to_json(ss, p);
```

```
iguana::xml::to_xml(ss, p);
iguana::msgpack::to_msgpack(ss, p);
```

Outline

- Concepts of reflection
- **Implementation of compile-time reflection**
- Application of compile-time reflection
- Prospect

Implementation of Compile-time Reflection

- Technology foundation
- Technical thought
- Concrete implementation
- Limitations
- Proposals of reflection

Technology Foundation

- C++11/14/17 features
  - variadic templates
  - std::tuple
  - auto lambda
  - std::apply
  - constexpr if
  - std::string_view
  - fold expression
  - ……
- Macros

Technology Foundation(Macros)

Count variadic arguments number

```
#define GET_ARG_COUNT(...)          GET_ARG_COUNT_INNER(__VA_ARGS__, RSEQ_N())
#define GET_ARG_COUNT_INNER(...)    ARG_N(__VA_ARGS__)
#define ARG_N(_1, _2, _3, _4, N, ...) N
#define RSEQ_N() 4, 3, 2, 1, 0

static_assert(GET_ARG_COUNT(a) == 1);
static_assert(GET_ARG_COUNT(a, b) == 2);
static_assert(GET_ARG_COUNT(a, b, c) == 3);
```

GET_ARG_COUNT(a) ➔ ARG_N(a, 4, 3, 2, **1**, 0)

ARG_N(_1, _2, _3, _4, **N**, ...) N ➔ 1

ARG_N(a, b, 4, 3, **2**, 1, 0)

ARG_N(_1, _2, _3, _4, **N**, ...) N ➔ 2

Technology Foundation(Macros)

```
#define RSEQ_N() 4,3,2,1,0
#define ARG_N(_1, _2, _3, _4, N, ...) N

#define GET_ARG_COUNT_INNER(...)     MARCO_EXPAND(ARG_N(__VA_ARGS__))
#define GET_ARG_COUNT(...)           GET_ARG_COUNT_INNER(__VA_ARGS__, RSEQ_N())

#define MARCO_EXPAND(x) x
```

MARCO_EXPAND for visual studio

Technology Foundation(Macros)

Connect string in ,

```
#define CON_STR_1(element, ...) #element
#define CON_STR_2(element, ...) #element SEPERATOR MARCO_EXPAND(CON_STR_1(__VA_ARGS__))
#define CON_STR_3(element, ...) #element SEPERATOR MARCO_EXPAND(CON_STR_2(__VA_ARGS__))
#define CON_STR_4(element, ...) #element SEPERATOR MARCO_EXPAND(CON_STR_3(__VA_ARGS__))

#define SEPERATOR ,

CON_STR_1(a)            "a"
CON_STR_2(a, b)         "a","b"
CON_STR_2(a, b, c)      "a","b",“c"
```

Technology Foundation(Macros)

Make an array

```
#define MAKE_ARRAY(NAME, ...) \
        MAKE_ARRAY_IMPL(NAME, GET_ARG_COUNT(__VA_ARGS__), __VA_ARGS__)

#define MAKE_ARRAY_IMPL(NAME, N, ...) \
constexpr std::array<const char*, N> arr_##NAME = { MACRO_CONCAT(CON_STR, N)(__VA_ARGS__) };

#define MACRO_CONCAT(A, B) A##_##B
```

MAKE_ARRAY(t, a, b)➔std::array<const char*, 2> arr_t = {"a","b"}

Technical Thought

Exist libraries about reflection

- magic_get
- boost.fusion
- boost.hana

Technical Thought

magic_get

```
struct foo {
    int some_integer;
    char c;
};


auto& r1 = boost::pfr::get<0>(f); //accessing field with index 0
auto& r2 = boost::pfr::get<1>(f);
```

T must be POD

the array will be flatted

Technical Thought

boost.fusion

```
struct person_t{
    std::string name;
    int age;
};

BOOST_FUSION_ADAPT_STRUCT(person_t,
    (std::string, name)
    (int, age)
);
```

```
person p = { "tom", 20 };
fusion::for_each(boost::mpl::range_c<unsigned, 0, fusion::result_of::size<person>::value>(),
    [&](auto index){
        std::cout << fusion::at_c<decltype(index)::value>(p)<<"\n";
    }
);
```

boost.fusion

- get value by index
- get name by index
- for_each fields

basic reflection functions

Technical Thought

boost.hana

```
struct person {
    std::string name;
    int age;
};

BOOST_HANA_ADAPT_STRUCT(not_my_namespace::person, name, age);

person john{ "John", 30 };
hana::for_each(john, [](auto pair) {
    std::cout << hana::to<char const*>(hana::first(pair)) << ":
"<< hana::second(pair) << std::endl;
});
```

```
#define BOOST_HANA_ADAPT_STRUCT_IMPL_17(TYPE , m1, m2, m3, m4, m5, m6, m7, m8, m9, m10, m11, m12, m13, m14, m15, m16)    \
    namespace boost { namespace hana {                                                                                  \
        template <>                                                                                                     \
        struct accessors_impl<TYPE> {                                                                                   \
            static constexpr auto apply() {                                                                             \
                struct member_names {                                                                                   \
                  static constexpr auto get() {                                                                         \
                      return ::boost::hana::make_tuple(                                                                 \
                          BOOST_HANA_PP_STRINGIZE(m1), BOOST_HANA_PP_STRINGIZE(m2), BOOST_HANA_PP_STRINGIZE(m3), BOOST_HANA_PP_STRINGIZE(m4), BOOST_HANA_PP_STRINGIZE(m5),
                          BOOST_HANA_PP_STRINGIZE(m6), BOOST_HANA_PP_STRINGIZE(m7), BOOST_HANA_PP_STRINGIZE(m8), BOOST_HANA_PP_STRINGIZE(m9), BOOST_HANA_PP_STRINGIZE(m10),
                          BOOST_HANA_PP_STRINGIZE(m11), BOOST_HANA_PP_STRINGIZE(m12), BOOST_HANA_PP_STRINGIZE(m13), BOOST_HANA_PP_STRINGIZE(m14), BOOST_HANA_PP_STRINGIZE(m15),
                          BOOST_HANA_PP_STRINGIZE(m16)    \
                      );                                                                                                \
                  }                                                                                                     \
                };                                                                                                      \
                return ::boost::hana::make_tuple(                                                                       \
                    ::boost::hana::make_pair(::boost::hana::struct_detail::prepare_member_name<0, member_names>(), ::boost::hana::struct_detail::member_ptr<decltype(&TYPE::m1),
                    &TYPE::m1>{}), ::boost::hana::make_pair(::boost::hana::struct_detail::prepare_member_name<1, member_names>(),
                    ::boost::hana::struct_detail::member_ptr<decltype(&TYPE::m2), &TYPE::m2>{}), ::boost::hana::make_pair(::boost::hana::struct_detail::prepare_member_name<2,
                    member_names>(), ::boost::hana::struct_detail::member_ptr<decltype(&TYPE::m3), &TYPE::m3>{}),
                    ::boost::hana::make_pair(::boost::hana::struct_detail::prepare_member_name<3, member_names>(), ::boost::hana::struct_detail::member_ptr<decltype(&TYPE::m4),
                    &TYPE::m4>{}), ::boost::hana::make_pair(::boost::hana::struct_detail::prepare_member_name<4, member_names>(),
                    ::boost::hana::struct_detail::member_ptr<decltype(&TYPE::m5), &TYPE::m5>{}), ::boost::hana::make_pair(::boost::hana::struct_detail::prepare_member_name<5,
                    member_names>(), ::boost::hana::struct_detail::member_ptr<decltype(&TYPE::m6), &TYPE::m6>{}),
                    ::boost::hana::make_pair(::boost::hana::struct_detail::prepare_member_name<6, member_names>(), ::boost::hana::struct_detail::member_ptr<decltype(&TYPE::m7),
                    &TYPE::m7>{}), ::boost::hana::make_pair(::boost::hana::struct_detail::prepare_member_name<7, member_names>(),
                    ::boost::hana::struct_detail::member_ptr<decltype(&TYPE::m8), &TYPE::m8>{}), ::boost::hana::make_pair(::boost::hana::struct_detail::prepare_member_name<8,
                    member_names>(), ::boost::hana::struct_detail::member_ptr<decltype(&TYPE::m9), &TYPE::m9>{}),
                    ::boost::hana::make_pair(::boost::hana::struct_detail::prepare_member_name<9, member_names>(), ::boost::hana::struct_detail::member_ptr<decltype(&TYPE::m10),
                    &TYPE::m10>{}), ::boost::hana::make_pair(::boost::hana::struct_detail::prepare_member_name<10, member_names>(),
                    ::boost::hana::struct_detail::member_ptr<decltype(&TYPE::m11), &TYPE::m11>{}), ::boost::hana::make_pair(::boost::hana::struct_detail::prepare_member_name<11,
                    member_names>(), ::boost::hana::struct_detail::member_ptr<decltype(&TYPE::m12), &TYPE::m12>{}),
                    ::boost::hana::make_pair(::boost::hana::struct_detail::prepare_member_name<12, member_names>(), ::boost::hana::struct_detail::member_ptr<decltype(&TYPE::m13),
                    &TYPE::m13>{}), ::boost::hana::make_pair(::boost::hana::struct_detail::prepare_member_name<13, member_names>(),
                    ::boost::hana::struct_detail::member_ptr<decltype(&TYPE::m14), &TYPE::m14>{}), ::boost::hana::make_pair(::boost::hana::struct_detail::prepare_member_name<14,
                    member_names>(), ::boost::hana::struct_detail::member_ptr<decltype(&TYPE::m15), &TYPE::m15>{}),
                    ::boost::hana::make_pair(::boost::hana::struct_detail::prepare_member_name<15, member_names>(), ::boost::hana::struct_detail::member_ptr<decltype(&TYPE::m16),
                    &TYPE::m16>{})\
                );                                                                                                      \
            }                                                                                                           \
        };                                                                                                              \
    }}                                                                                                                  \
```

Technical Thought

- metadata defination
- operations of metadata
  - get each field
  - for_each every field

- no limitation about the reflection object
- non-intrusive

Technical Thought

- A simple implementation

```
//the field information: field type, field value, field index
template<typename T, T mPtr, unsigned Index>
struct MemberBinding
{
    constexpr static T value = mPtr;
    constexpr static unsigned index = Index;
};

//pack all fields(MemberBinding)
template<typename...> struct Pack {};
```

```cpp
struct Aggregate {
    int member1;
    std::string member2;
};
template<typename> struct Members {};
template<> struct Members<Aggregate> {
    using type = Pack<
        MemberBinding<decltype(&Aggregate::member1), &Aggregate::member1, 0 >,
        MemberBinding<decltype(&Aggregate::member2), &Aggregate::member2, 1 >
    >;

    constexpr static const char *name = "Aggregate";

    static const char *const *names() {
        static const char *rv[] = { "member1", "member2" };
        return rv;
    }
};
```

object type

fields type

fields value

fields index

fields name

```
template<typename A, typename MT, MT MPTR, unsigned Ndx, typename... Rest>
void printPack(std::ostream &out, const A &v,
                        Pack<MemberBinding<MT, MPTR, Ndx>, Rest...> *)
{
    using M = Members<A>;
    out << ' ' << Ndx << ':' << M::names()[Ndx] << ':' << v.*MPTR;
    printPack(out, v, (Pack<Rest...> *)nullptr);
}

template<typename A>
void printPack(std::ostream &out, const A &v, Pack<> *) {}


using M = Members<T>;
printPack(ostr, v, (typename M::type *)nullptr);
```

all fields information

cppcon2016: Achieving financial data processing performance through compile time introspection

Technical Thought

- The specialized template class saves the object type
- Variadic template class Pack<…> save all fields information

```
template<typename T,  T mPtr,  unsigned Index>
struct MemberBinding;
```

- A string array save all the names of fields
- Visit all fields by recursively foreach pack<…>

```cpp
struct Aggregate {
    int member1;
    std::string member2;
};
```

- define metadata is quite overelaborate
- need handwritten, can't automatic
- reduplicate for the other reflection objects

define metadata

```cpp
template<> struct Members<Aggregate> {
    using type = Pack<
        MemberBinding<decltype(&Aggregate::member1), &Aggregate::member1, 0  >,
        MemberBinding<decltype(&Aggregate::member2), &Aggregate::member2, 1  >
    >;

    constexpr static const char *name = "Aggregate";

    static const char *const *names() {
        static const char *rv[] = { "member1", "member2" };
        return rv;
    }
};
```

Members<Aggregate>, Members<Person>, Members<Other>,……

how to automaticly generate the metadata of an arbitray object?

How to Automatically Generate The Metadata of An Arbitray Object?

- Automatically pack all fields by macros and new features
- Automatically create a field name array by macros and new features
- Provide a generic for_each algorithm

```
#define MAKE_META_DATA(STRUCT_NAME, N, ...) \
    constexpr inline std::array<std::string_view, N> arr_##STRUCT_NAME =
        { MARCO_EXPAND(MACRO_CONCAT(CON_STR, N)(__VA_ARGS__)) };\
    MAKE_META_DATA_IMPL(STRUCT_NAME, MAKE_ARG_LIST(N, &STRUCT_NAME::OBJECT, __VA_ARGS__))
```

automatically pack all fields

automatically create a std::array<std::string_view, N> of fields names

```cpp
template<typename> struct Members {};

#define MAKE_META_DATA_IMPL(STRUCT_NAME, ...)\
template<>struct Members<STRUCT_NAME>{\
    constexpr decltype(auto) static apply_impl(){\
            return std::make_tuple(__VA_ARGS__);\
    }\
    using type = void; \
    using size_type = std::integral_constant<size_t, GET_ARG_COUNT(__VA_ARGS__)>;\
    constexpr static size_t value() { return size_type::value; }\
    constexpr static std::string_view name() {\
        return std::string_view(#STRUCT_NAME, sizeof(#STRUCT_NAME));\
    }\
    constexpr static std::array<std::string_view, size_type::value> arr() {\
        return arr_##STRUCT_NAME;\
    }\
};

#define MAKE_META_DATA(STRUCT_NAME, N, ...) \
    constexpr inline std::array<std::string_view, N> arr_##STRUCT_NAME =
        { MARCO_EXPAND(MACRO_CONCAT(CON_STR, N)(__VA_ARGS__)) };\
    MAKE_META_DATA_IMPL(STRUCT_NAME, MAKE_ARG_LIST(N, &STRUCT_NAME::FIELD, __VA_ARGS__))
```

```
#define MAKE_META_DATA(STRUCT_NAME, N, ...) \
MAKE_META_DATA_IMPL(STRUCT_NAME, MAKE_ARG_LIST(N, &STRUCT_NAME::OBJECT, __VA_ARGS__))
```

```
(&SomeObject::field1, &SomeObject::field2, &SomeObject::field3,....., &SomeObject::fieldN )
```

```
#define MAKE_META_DATA_IMPL(STRUCT_NAME, ...)\
template<>struct Members<STRUCT_NAME>{\
    constexpr decltype(auto) static apply_impl(){\
            return std::make_tuple(__VA_ARGS__);\
    }\
```

```
#define REFLECTION(STRUCT_NAME, ...) \
MAKE_META_DATA(STRUCT_NAME, GET_ARG_COUNT(__VA_ARGS__), __VA_ARGS__)
```

automatically pack all member variables

```
/* arg list expand macro, now support 120 args */
#define MAKE_ARG_LIST_1(op, arg, ...)   op(arg)
#define MAKE_ARG_LIST_2(op, arg, ...)   op(arg), MARCO_EXPAND(MAKE_ARG_LIST_1(op, __VA_ARGS__))
#define MAKE_ARG_LIST_3(op, arg, ...)   op(arg), MARCO_EXPAND(MAKE_ARG_LIST_2(op, __VA_ARGS__))
#define MAKE_ARG_LIST_4(op, arg, ...)   op(arg), MARCO_EXPAND(MAKE_ARG_LIST_3(op, __VA_ARGS__))
#define MAKE_ARG_LIST_5(op, arg, ...)   op(arg), MARCO_EXPAND(MAKE_ARG_LIST_4(op, __VA_ARGS__))
#define MAKE_ARG_LIST_6(op, arg, ...)   op(arg), MARCO_EXPAND(MAKE_ARG_LIST_5(op, __VA_ARGS__))
#define MAKE_ARG_LIST_7(op, arg, ...)   op(arg), MARCO_EXPAND(MAKE_ARG_LIST_6(op, __VA_ARGS__))
#define MAKE_ARG_LIST_8(op, arg, ...)   op(arg), MARCO_EXPAND(MAKE_ARG_LIST_7(op, __VA_ARGS__))
#define MAKE_ARG_LIST_9(op, arg, ...)   op(arg), MARCO_EXPAND(MAKE_ARG_LIST_8(op, __VA_ARGS__))
#define MAKE_ARG_LIST_10(op, arg, ...)  op(arg), MARCO_EXPAND(MAKE_ARG_LIST_9(op, __VA_ARGS__))
#define MAKE_ARG_LIST_11(op, arg, ...)  op(arg), MARCO_EXPAND(MAKE_ARG_LIST_10(op, __VA_ARGS__))
#define MAKE_ARG_LIST_12(op, arg, ...)  op(arg), MARCO_EXPAND(MAKE_ARG_LIST_11(op, __VA_ARGS__))
#define MAKE_ARG_LIST_13(op, arg, ...)  op(arg), MARCO_EXPAND(MAKE_ARG_LIST_12(op, __VA_ARGS__))
#define MAKE_ARG_LIST_14(op, arg, ...)  op(arg), MARCO_EXPAND(MAKE_ARG_LIST_13(op, __VA_ARGS__))
#define MAKE_ARG_LIST_15(op, arg, ...)  op(arg), MARCO_EXPAND(MAKE_ARG_LIST_14(op, __VA_ARGS__))
#define MAKE_ARG_LIST_16(op, arg, ...)  op(arg), MARCO_EXPAND(MAKE_ARG_LIST_15(op, __VA_ARGS__))
#define MAKE_ARG_LIST_17(op, arg, ...)  op(arg), MARCO_EXPAND(MAKE_ARG_LIST_16(op, __VA_ARGS__))
#define MAKE_ARG_LIST_18(op, arg, ...)  op(arg), MARCO_EXPAND(MAKE_ARG_LIST_17(op, __VA_ARGS__))
#define MAKE_ARG_LIST_19(op, arg, ...)  op(arg), MARCO_EXPAND(MAKE_ARG_LIST_18(op, __VA_ARGS__))
#define MAKE_ARG_LIST_20(op, arg, ...)  op(arg), MARCO_EXPAND(MAKE_ARG_LIST_19(op, __VA_ARGS__))
#define MAKE_ARG_LIST_21(op, arg, ...)  op(arg), MARCO_EXPAND(MAKE_ARG_LIST_20(op, __VA_ARGS__))
#define MAKE_ARG_LIST_22(op, arg, ...)  op(arg), MARCO_EXPAND(MAKE_ARG_LIST_21(op, __VA_ARGS__))
#define MAKE_ARG_LIST_23(op, arg, ...)  op(arg), MARCO_EXPAND(MAKE_ARG_LIST_22(op, __VA_ARGS__))

#define MAKE_ARG_LIST_118(op, arg, ...)  op(arg), MARCO_EXPAND(MAKE_ARG_LIST_117(op, __VA_ARGS__))
#define MAKE_ARG_LIST_119(op, arg, ...)  op(arg), MARCO_EXPAND(MAKE_ARG_LIST_118(op, __VA_ARGS__))
#define MAKE_ARG_LIST_120(op, arg, ...)  op(arg), MARCO_EXPAND(MAKE_ARG_LIST_119(op, __VA_ARGS__))
```

```
struct person
{
        std::string     name;
        int64_t          age;
};

REFLECTION(person, name, age);
```

```
template<typename...> struct Pack {};
template<typename> struct Members {};
template<> struct Members<Person> {
    using type = Pack<
        MemberBinding<decltype(&Person::name), &Person::name, 0  >,
        MemberBinding<decltype(&Person::age), &Person::age, 1  >
    >;
    static const size_t value() { return 2; }
    static const char *name = "Person";
    static const char *const *names() {
    static const char *rv[] = { "name", "age" };
        return rv;
    }
};
```

automatically define metadata

```
template <class T>
constexpr auto tie_as_tuple(T&& val, size_t_<1>) noexcept {
   auto&[a] = std::forward<T>(val);
   return make_tuple_of_references(a);
}

template <class T>
constexpr auto tie_as_tuple(T&& val, size_t_<2>) noexcept {
   auto&[a, b] = std::forward<T>(val);
   return make_tuple_of_references(a, b);
}

template <class T>
constexpr auto tie_as_tuple(T&& val, size_t_<12>) noexcept {
   auto&[a, b, c, d, e, f, g, h, j, k, l, m] = std::forward<T>(val);
   return make_tuple_of_references(a, b, c, d, e, f, g, h, j, k, l, m);
}
```

Operations of Metadata

```
template<typename T>
std::enable_if_t<is_reflection<T>::value, size_t> get_value()
{
    using M = Members<std::remove_const_t <std::remove_reference_t<T>>>;
    return M::value();
}

template <typename T, typename = void>
struct is_reflection : std::false_type{};

template <typename T>
struct is_reflection<T, void_t<
        typename Members<std::remove_const_t <std::remove_reference_t<T>>>::type
        >> : std::true_type{
};
```

```cpp
template<typename T, size_t  I>
constexpr const std::string_view get_name()
{
    using M = Members<std::remove_const_t <std::remove_reference_t<T>>>;
    static_assert(I<M::value(), "out of range");

    return M::arr()[I];
}
```

```cpp
//get the field by index
template<size_t I, typename T>
constexpr decltype(auto) get(T&& t)
{
    using M = Members<std::remove_const_t <std::remove_reference_t<T>>>;
    static_assert(I<M::value(), "out of range");

    return std::forward<T>(t).*(std::get<I>(M::apply_impl()));
}
```

for_each Metadata

```
template<typename T, typename F>
constexpr std::enable_if_t<is_reflection<T>::value> for_each(T&& t, F&& f)
{
    using M = Members<std::remove_const_t <std::remove_reference_t<T>>>;
    for_each(M::apply_impl(), std::forward<F>(f), std::make_index_sequence<M::value()>{});
}

template <typename... Args, typename F, std::size_t... Idx>
constexpr void for_each(const std::tuple<Args...>& t, F&& f, std::index_sequence<Idx...>)
{
    (std::forward<F>(f)(std::get<Idx>(t), std::integral_constant<size_t, Idx>{}), ...);
}
```

```cpp
template <typename... Args, typename Func, std::size_t... Idx>
void for_each(const std::tuple<Args...>& t, Func&& f, std::index_sequence<Idx...>) {
    (void)std::initializer_list<int> { (f(std::get<Idx>(t)), void(), 0)...};
}

template <typename... Args, typename Func, std::size_t... Idx>
void for_each(const std::tuple<Args...>& t, Func&& f, std::index_sequence<Idx...>) {
    (f(std::get<Idx>(t)), ...);
}
```

fold expression

```
struct Person {
    std::string name;
    int age;
};
REFLECTION(Person, name, age)


std::cout << get_name<Person, 0>() << std::endl;
std::cout << get_name<Person, 1>() << std::endl;
std::cout << get<0>(p) << std::endl;
std::cout << get<1>(p) << std::endl;

Person p = { "admin", 20 };
for_each(p, [](const auto& item, auto idx) {
    std::cout << item << " " << index << " " << decltype(i)::value << std::endl;
});
```

https://github.com/qicosmos/iguana

Limitations

- Can’t reflect member functions
- Can’t reflect private members

Proposals

- Reflection
  - R194r3 by Matúš Chochlík and Axel Naumann
  - P0670r0 by Chochlík, Naumann and David Sankel

```
template <Record T> struct get_public_data_members;
template <Record T> struct get_accessible_data_members;
template <Record T> struct get_data_members;

template <Record T> struct get_public_member_functions;
template <Record T> struct get_accessible_member_functions;
template <Record T> struct get_member_functions;
```

Proposals

- Metaclass
  - P0707r1 by Herb Sutter
  - Based on static reflection, next-level layer of abstraction

```
constexpr {                                  // execute this at compile time
    for... (auto m : $T.variables())         // examine each member variable m in T
        if (m.name() == “xyzzy”)             // if there is one with name “xyzzy”
            -> { int plugh; }                // then inject also an int named “plugh”
}

constexpr {
    for... (auto f : $T.functions())         // for each member function f in T
        if (f.is_copy() || f.is_move())      // let’s say we want to disallow copy/move
            compiler.error("this type may not have a copy or move function", f);
}           // note: passing f will use f.source_location() for this diagnostic message
```

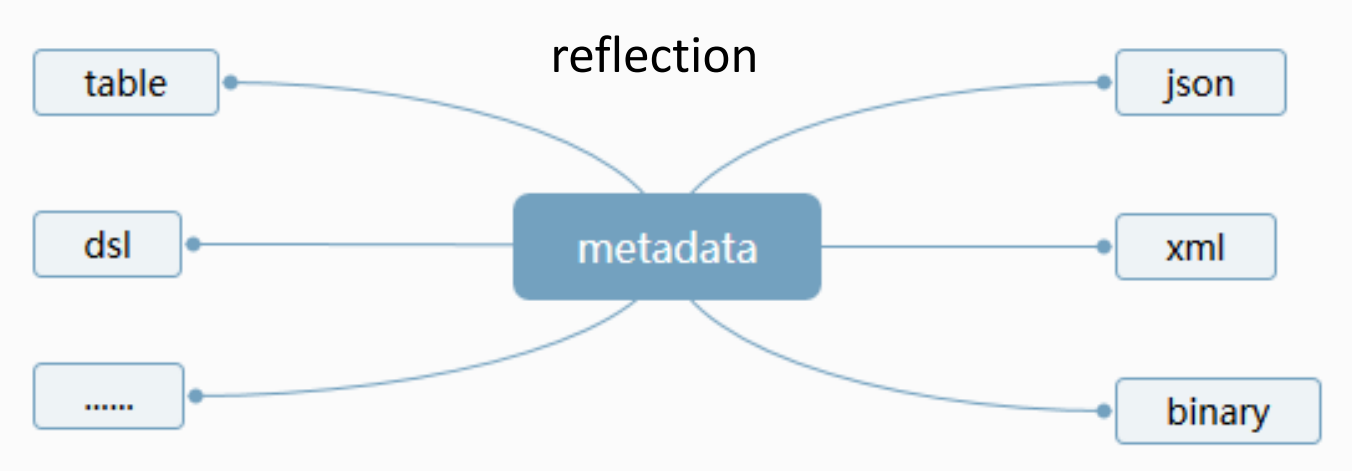
Outline

- Concepts of reflection
- Implementation of compile-time reflection
- **Application of compile-time reflection**
- Prospect

Application of Compile-time Reflection

- Serialization engine
- ORM(Object Relational Mapping)

Serialization Engine



```
struct person
{
        std::string     name;
        int64_t          age;
};

REFLECTION(person, name, age);
```

```
person p = { "admin", 20 };
iguana::string_stream ss;

iguana::json::to_json(ss, p);
iguana::xml::to_xml(ss, p);

person p2;
iguana::json::from_json(ss.str(), p2);
iguana::json::from_xml(ss.str(), p2);
```

https://github.com/qicosmos/iguana

```
constexpr auto to_json(Stream& s, T &&t) -> std::enable_if_t<is_reflection<T>::value> {
        s.put('{');
        for_each(std::forward<T>(t), [&t, &s](const auto &v, auto i) {
            using M = decltype(iguana_reflect_members(std::forward<T>(t)));
            constexpr auto Idx = decltype(i)::value;
            constexpr auto Count = M::value();

            write_json_key(s, i, t);
            s.put(':');

            if constexpr (!is_reflection<decltype(v)>::value) {
                render_json_value(s, t.*v);
            }
            else {
                to_json(s, t.*v);
            }

            if (Idx < Count - 1)
                s.put(',');
        });

        s.put('}');
};
```

for the nested struct

```
constexpr auto write_json_key = [](auto& s, auto i, auto& t) {
    s.put('"');
    auto name = get_name<decltype(t), decltype(i)::value>();

    s.write(name.data(), name.length() - 1);
    s.put('"');
};
```

```cpp
template<typename T, typename = std::enable_if_t<is_reflection<T>::value>>
constexpr void do_read(xml_reader_t &rd, T &&t){
    for_each(std::forward<T>(t), [&t, &rd](const auto v, auto i){
        if constexpr (!is_reflection<decltype(t.*v)>::value){
            if (rd.begin_object(get_name<T, Idx>().data()) == object_status::NORMAL){
                rd.get_value(t.*v);

                rd.end_object(get_name<T, Idx>().data());
            }
        }
        else{
            if (rd.begin_object(get_name<T, Idx>().data()) == object_status::NORMAL){
                do_read(rd, t.*v);
                rd.end_object(get_name<T, Idx>().data());
            }
        }
    });
}
```

```cpp
template<typename T>
std::enable_if_t<std::is_integral<T>::value, std::string> to_str(T t)
{
    return std::to_string(t);
}

template<typename T>
std::enable_if_t<!std::is_integral<T>::value, std::string> to_str(T t)
{
    return t;
}

template<typename T>
auto to_str17(T t)
{
    if constexpr(std::is_integral<T>::value)
        return std::to_string(t);
    else
        return t;
}
```

ORM(Object Relational Mapping)

```
struct user{
    int id;
    std::string name;
    std::string pwd;
    std::string qq;
    int sex;
    int role;
};
REFLECTION(user, id, name, pwd, qq, sex, role);

const char* get_all1 = "select * from user;";
std::vector<user> users = db.query<user>(get_all);

const char* get_all2 = "select user.*, article.title, contact.author_id from
        user,article,contact;";
std::vector<std::tuple<user, std::string, int>> parts = db.query<user, std::string,
        int>(get_all1);
```

https://github.com/qicosmos/ormpp

```
std::vector<T> v;
while (true){
    result = sqlite3_step(stmt_);

    T t;
    for_each(t, [this](auto& item, auto I){
        constexpr auto index = decltype(I)::value;
        if (SQLITE_NULL == sqlite3_column_type(stmt_, index)){
            return;
        }

        sqlite::assign(stmt_, item);
    });

    v.push_back(std::move(t));
}
```

```cpp
template<typename T>
std::string make_insert_sql(){
    std::string str_atr = "insert into ";
    std::string str = "values(";
    using M = Members<std::remove_const_t <std::remove_reference_t<T>>>;
    str_atr += M::name();
    str_atr += +"(";
    const int size = M::value();
    for (size_t i = 0; i < size; i++){
        str_atr += get_name<T>(i);
        str += "?";
        ...
    }

    return str_atr + str;
}
```

```cpp
db.make_insert_sql<user>();
```

```
//insert into user(id, name,pwd,qq,sex,role) values(?,?,?,?,?,?);
```

Outline

- Concepts of reflection
- Implementation of compile-time reflection
- Application of compile-time reflection
- **Prospect**

Prospect

- DSL(domain-specific language)
- Data binding
- Protocols adaptor

DSL

```
struct Person {
    std::string name;
    int age;
};
REFLECTION(Person, name, age)
```

Person.cs

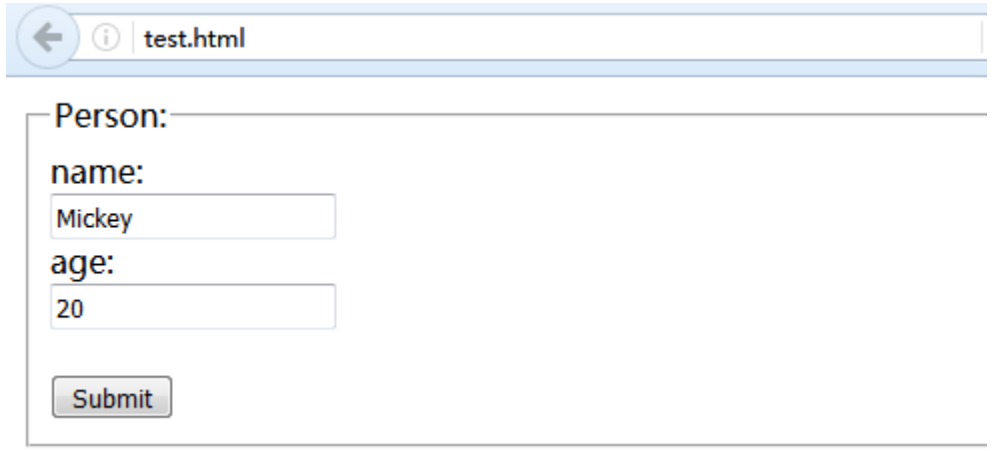
```
class Person{
    public string name;
    public int age;
}
```

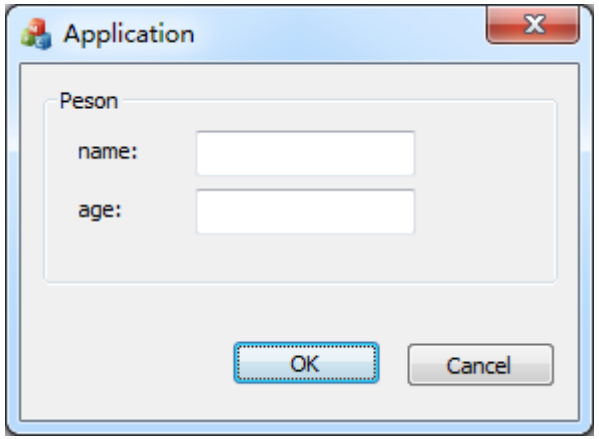
Person.java

```
class Person {
    String name;
    int age;
}
```

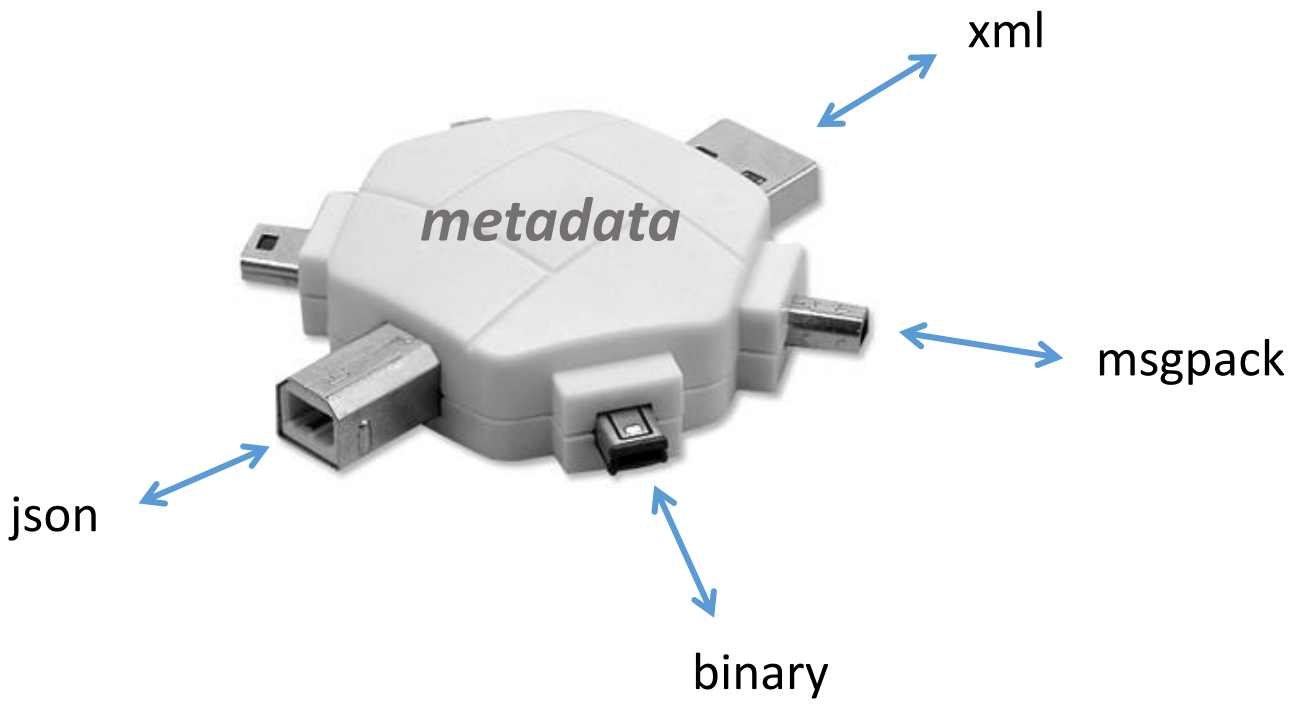
Data Binding

```
struct Person {
    std::string name;
    int age;
};
REFLECTION(Person, name, age)
```





Protocols Adaptor



Questions?