P0194 – *Static reflection*

Matúš Chochlík
Axel Naumann

Call for Compile-Time Reflection Proposals – **N3814**

- Targeted use cases:
  - **Generation of common functions**
    - comparison operators,
    - serialization functions,
    - hash functions,
    - ...
  - **Type transformations**
    - *struct-of-arrays*,
    - object-relational mapping,
    - ...
  - **Compile-time context information**
    - replacement for `__FILE__`, `__LINE__`, ...
  - **Enumeration of other entities**
    - types in a namespace
    - parameters of a function
    - global variables in a namespace
    - enumerators in an enumeration
    - ...

Static reflection proposal

- **Current iteration**
  - **P0194R1** – *Static reflection (rev.5)* – wording
  - **P0385R0** – *Static reflection: Rationale, design and evolution*
- Previous revisions[1]
  - **P0194R0** – *Static reflection (rev.4)*
  - **N4451** – *Static reflection (rev.3)*
  - **N4452** – *A case for strong static reflection*
  - **N4111** – *Static reflection (rev.2)*
  - **P3996** – *Static reflection (rev.1)*

[1] see the revision history in P0194R1

Metaobject – abstract definition

- **A representation of a base-level program declaration.**
- **Gives a first-class identity to the reflected entity.**
  - Can be stored in a "variable".
  - Can be passed as an argument or a return value in a metaprogram.
  - Separates the reflection of a declaration from the querying of the metadata.
- **Can be reasoned-about at compile-time.**
- In some cases allows to "go back" to the reflected declaration.
- Conforms to one of the *metaobject concepts*.

Metaobjects – implementation

- **Anonymous, implementation-defined types.**
  - have first-class identity at compile-time
  - lightweight
  - no visible internal structure
  - creation of run-time objects of these types is not required
  - can be implemented as template-wrapped constants
- Returned by the **reflexpr** operator.

```
using meta_global_scope = reflexpr(::);
using meta_std = reflexpr(std);
using meta_int = reflexpr(int);
```

Metaobject kinds

- **In the current proposal**
  - the global scope,
  - a namespace,
  - a type,
  - a class,
  - a class data member,
  - an enumeration,
  - an enumeration value,
  - a namespace or type alias,
  - a specifier.
- Future extensions[2]
  - a template,
  - a variable
  - a function,
  - a constructor,
  - an operator,
  - a template parameter,
  - a function parameter,
  - ...

[2] Using the same principles. See also N4111

Metaobject operations

- **Compile-time operations adding functionality to the metaobjects.**
- One or several of their arguments are *metaobjects*.
- Return compile-time metadata or other metaobjects.
- **Implemented as class templates:**

```
using meta_str = reflexpr(std::string);
get_name_v<meta_str>;              // "string"
using meta_std = get_scope_m<meta_str>;
get_name_v<meta_std>;              // "std"
reflected_type_t<meta_str> str("blah");
```

Metaobject operations – "return values"

- **Compile-time metadata** in the form of:
  - boolean constants,
  - integral constants,
  - enumerator constants,
  - string constants,
  - other (data member pointers, . . .)
- **Metaobjects** reflecting:
  - scope,
  - class data members, class member typedefs, class inheritance,
  - parameters, the return value,
  - aliased declarations,
  - various specifiers,
  - . . .

Metaobject concepts

- Determine the **category** of a metaobject – **what does it reflect**.
- **Determine which operations can be invoked on a metaobject.**

  - `Object`
  - `ObjectSequence`
  - `Reversible`
  - `Named`
  - `Typed`
  - `ScopeMember`
  - `Scope`
  - `Alias`
  - `ClassMember`
  - `Linkable`
  - `Constant`
  - `Specifier`
  - `Namespace`
  - `GlobalScope`
  - `NamespaceAlias`
  - `Type`
  - `TypeAlias`
  - `Class`
  - `Enum`
  - `EnumClass`
  - `Variable`
  - `DataMember`
  - `MemberType`
  - `EnumValue`

Metaobject concepts

- **Distinguishing metaobjects** from other types – **the** `is_metaobject` **type trait**:

```
template <typename T> struct is_metaobject
 : integral_constant<bool, implementation-defined> { };

template <typename T> constexpr bool is_metaobject_v
 = is_metaobject<T>::value;
```

- **The** `meta::Object` **concept:**

```
template <typename T> concept bool Object
 = is_metaobject_v<T>;
```

- All other **metaobjects form a** generalization-specialization **hierarchy** and have additional requirements:

```
template <typename T> concept bool Named
 = Object<T> && implementation-defined;
```

Metaobject sequences

- Lightweight "handle" to an ordered sequence of metaobjects.
  - class or namespace members,
  - list of storage class specifiers,
  - list of base class inheritance specifiers,
  - ...
- **Does not instantiate "contained" metaobjects eagerly.**
- Definition:

```
template <typename T> concept bool ObjectSequence
 = Object<T> && implementation-defined;
```

- **Operations:**

```
template <ObjectSequence S>           struct get_size;
template <ObjectSequence S, size_t I> struct get_element;

template <ObjectSequence S, template <class...> class Tpl>
struct unpack_sequence;
```

The <reflexpr> header file

- **Must be included** prior to the use of `reflexpr`.
- Defines the metaobject concepts.
- Implements the metaobject operations.
- All **reflection-related declarations go to** the **namespace** `std::meta`, **except for** the `is_metaobject` **trait**:

```
namespace std {
  template <typename T> is_metaobject;

  namespace meta {
    template <typename T> concept bool Object;
    template <Object T> concept bool Named;
    ...
    template <Named MO> struct get_name;
    ...
  } // namespace meta
} // namespace std
```

Why a template metaprogramming interface?

- **Tried paradigm** with approximately 15 years of experience.
- The pros and cons are generally known.
- Lots of **existing TMP libraries** and utilities.
- **Consistent with** the existing **type-traits** and other standard and third-party metaprogramming utilities.
- **Other interfaces**, both compile-time and run-time **can** and will **be implemented on top of**[3] **the TMP interface**.

```
template <Object MO> struct metaobject;

template <Object MO>
constexpr auto get_name(metaobject<MO>) {
    return meta::get_name_v<MO>;
}
```

[3] or parallel to, using the same machinery

Design – the good

- All the "magic" is contained within the `reflexpr` operator.
- No changes to the core language required[4].
- Metaobjects are first-class entities.
- Allows for lightweight and efficient implementation.
- Fairly powerful and expressive.
- Covers many use cases.
- Non-intrusive.
- Fine-grained.
- Extensible.
- Lazy[5].

[4] except for the operator
[5] can be a virtue too

Design – the bad

- Requires a new keyword – `reflexpr`.
  - We did some checking – “reflexpr” is not a very common word.
  - Zero occurrences in ACTCD16[6]

[6] http://www.tomazos.com/actcd16/

Design – the ugly

- It's template metaprogramming[7]
  - May be too verbose for some use cases – can[8] be solved by implementing a simplifying façade.
- **But, we don't want to trade "simplicity" for usefulness.**

[7] for the moment ...
[8] and will

Typedef reflection – unnamed struct

- Suppose that we have the following `typedef` to an unnamed `struct`[9]:

```
typedef struct { int a, b, c; } my_struct;
```

- **We want to serialize instances** of this `struct` for network transport, **including the type name** in the data format:

```
cout << "{" << '"' << "type" << '"' << ":␣" << '"'
     << get_name_v<reflexpr(my_struct)>
     << '"' << ",␣" << '"' << "attr" << '"' << ":␣{";
/* ... */
cout << "}}";
```

- Output **without** `typedef` reflection:

```
{"type":          "", "attr": { ... }}
```

- Output **with** `typedef` reflection:

```
{"type": "my_struct", "attr": { ... }}
```

[9] possibly in third-party code

Typedef reflection – inter-platform differences

- Let's suppose that we have the following declarations:

```
#if PLATFORM_X
  typedef int         int32_t;
  typedef long        int64_t;
#elif PLATFORM_Y
  typedef long        int32_t;
  typedef long long   int64_t;
#endif
struct abc { int64_t a, b, c; };
```

- We serialize instances of this `struct` **on platform X** and send them over the network to platform Y:
- **Without** `typedef` reflection:

```
"attr":{"a":{"t":"long", "v":"9223372036854775 8"},...}
```

- **With** `typedef` reflection:

```
"attr":{"a":{"t":"int64_t","v":"9223372036854775 8"},...}
```

Typedef reflection – inter-platform differences (cont.)

- . . . meanwhile at the receiving end **on platform Y**:
  - **Without** `typedef` reflection

```
json_obj obj = json_obj::from_string(json_str);
auto attr = obj["attr"]["a"];
/* ... */
if(attr["t"] == "long") {
   long tmp;
   attr["v"] >> tmp; // 32-bit int overflow?, exception?
} /* ... */
```

  - **With** `typedef` reflection

```
json_obj obj = json_obj::from_string(json_str);
auto attr = obj["attr"]["a"];
/* ... */
if(attr["t"] == "int64_t") {
   int64_t tmp;
   attr["v"] >> tmp; // A-OK
} /* ... */
```

Typedef reflection – Diagnostics and logging

- We want to log function calls – why not **use reflection to get type names**, source location info, etc.
- Output **without** `typedef` reflection:
  - Depends on the platform, compiler and the implementation of the standard library.
  - May depend on build configuration (debug/release, 32-bit/64-bit)
- Output **with** `typedef` reflection:
  - Same on every platform
  - Same for every build configuration
  - Preserves semantic meaning
  - Matches the names in output to what is used in the code.

Compiler diagnostics use typedefs for a good reason. Same holds for reflection.

Non-public member reflection

- **Necessary for several use cases** (serialization, etc.)
- Guidance from JAX: *Non-public reflection yes, but* greppable.
- **Variants of operations** for *obtaining* the reflections of the non-public members:
  - `get_data_members` vs. `get_all_data_members`
  - `get_member_types` vs. `get_all_member_types`
  - ...
- Variants of operations for *going back* to the base-level?[10]
  - `get_pointer` vs. `get_nonpublic_pointer`
  - `get_reflected_type` vs. `get_nonpublic_reflected_type`
  - *greppable*, but does not scale very well
- Freely access metadata; name, type information, scope information, etc.

[10] not in the proposal at the moment

Reverse reflection

- **What happens when a metaobject is passed as the argument to** `reflexpr`?
  - If the metaobject is not a model of `Reversible` – ill-formed.
  - If the metaobject is a `Reversible` – *reverse reflection*.
- **Reverse reflection** – as if the reflected declaration was used:
  - **Already in the proposal**:

```
using meta_int = reflexpr(int);
// same as: int i = 123;
reflexpr(meta_int) i = 123;
// equivalent to:
get_reflected_type_t<meta_int> i = 123;
```

  - Extensions of this mechanism are planned for the future.

Using the reflexpr operator

Valid operands for the `reflexpr` operator:

```
reflexpr() // reflects the global namespace
reflexpr(::) // reflects the global namespace
reflexpr(std) // reflects the namespace
reflexpr(unsigned) // reflects the type
reflexpr(std::size_t) // reflects the typedef
reflexpr(std::launch) // reflects the enum type
reflexpr(std::vector<int>) // reflects the class
```

Some of these **will become valid** in the future:

```
reflexpr(1) // ill-formed
reflexpr(std::sin) // ill-formed
reflexpr(std::vector) // ill-formed
reflexpr(is_same_v<void,void>) // ill-formed
```

Getting class member types

```
struct foo {
  int a;
  bool b;
  char c;
};

template <meta::DataMember ... MDM> using helper
 = tuple<meta::get_reflected_type_t<MDM>...>;

using X = meta::unpack_sequence_t<
  meta::get_data_members<reflexpr(foo)>, helper
>;

is_same_v<X, tuple<int,bool,char>>; // true
```

Getting class member names

```
struct foo { int a; bool b; char c; double d; };

template <meta::DataMember...MDM> struct helper {
  static const char** get(void) {
    static const char* n[] = {
      meta::get_name_v<MDM>...
    };
    return n;
  };
};
template <typename T>
using name_getter = meta::unpack_sequence_t<
  meta::get_data_members<reflexpr(T)>, helper
>;

name_getter<foo>::get()[0]; // "a"
```

Unnamed named

- What about this:

```
struct outer {
  struct {
    int i;
  } inner;
};
using meta_X = get_type_m<reflexpr(outer.inner)>;
```

- **A** – `meta_X` **is not a** `meta::Named`
  - `meta::get_name_v<meta_X>` is ill-formed
  - Easy to detect unnamed declarations
  - May complicate other things
- **B** – `meta_X` **is a** `meta::Named`
  - `meta::get_name_v<meta_X>` returns `""`
  - More clumsy to detect unnamed declarations
    - Add a new operation – `bool(meta::is_unnamed<X>::value)`?

New implementation in `clang`

- **New fundamental type** `__metaobject_id` – basically a ripped-off of `uintptr_t`.
- **New operator** `__reflexpr` – same arguments as `reflexpr`, returns `__metaobject_id`.
- A metaobject is implemented as:

```
template <__metaobject_id MoId>
struct __metaobject;
```

- **Operations implemented by built-ins:**

```
constexpr __metaobject_id
__metaobj_get_scope(__metaobject_id);
```

- **Wrapped into templates:**

```
template <typename MO>
struct get_scope;

template <__metaobject_id MoId>
struct get_scope<__metaobject<MoId>> {
    typedef __metaobject<
        __metaobj_get_scope(MoId)
    > type;
};
```

DISCLAIMER:

The following are **future extensions**, not part of this proposal. We want to lay the foundation before we paint the walls!

Reverse reflection 2.0

- We already know that ...

```
using meta_int = reflexpr(int);
// same as: int i = 123;
reflexpr(meta_int) i = 123;
```

- what if ...

```
struct S { int i; };
S s{123};
using meta_S_i = reflexpr(S::i);
// same as:  assert(s.i == 123);
assert(s.reflexpr(meta_S_i) == 123);

using meta_std = reflexpr(std);
// same as: std::string str;
reflexpr(meta_std)::string str;

using meta_std_pair = reflexpr(std::pair);
//          std::pair<int, int> pii;
reflexpr(meta_std_pair)<int, int> pii;
```

Identifier formatting

- The **ability to generate identifiers programmatically**[11] is important for several use cases.
  - Structure-of-arrays generators
  - Object-relational mapping
  - ...
- Let's use special operator, say `identifier`[12], with a **format string literal and a set of** `meta::Named` **metaobjects**:

```
struct bar { int foo; };
using m_bar = reflexpr(bar);
using m_foo = reflexpr(bar::foo);

int identifier("baz");                      //int baz;
int identifier("%1", m_bar);                //int bar;
int identifier("%2_%1", m_bar, m_foo);      //int foo_bar;
int identifier("my_%1_%2", m_bar, m_foo);//int my_bar_foo;
```

[11] without the preprocessor

[12] let the bike-shedding begin

Variadic composition

- suppose we have a struct which we want to transform;

```
struct original { T1 attr1; T2 attr2; T3 attr3; };
```

- by using identifier generation and variadic composition,

```
template <DataMember ... MDM> struct soa_helper {
    std::vector<reflexpr(meta::get_type_t<MDM>)>
        identifier("vec_%1", MDM)...;
};
```

- instantiating the soa_helper:

```
using mdm = get_data_members_m<reflexpr(original)>;
using soa_orig = expand_sequence_t<mdm, soa_helper>;
```

- would result in the following structure:

```
struct {
    std::vector<T1> vec_attr1;
    std::vector<T2> vec_attr2;
    std::vector<T3> vec_attr3;
};
```

Context-dependent reflection

- Allows to **obtain metadata based on the context in which the reflection operator is invoked**, instead of on the declaration name:
  - `reflexpr(this::namespace)` – the innermost enclosing namespace.
  - `reflexpr(this::class)` – the innermost enclosing class.
  - `reflexpr(this::template)` – the innermost enclosing template.
  - `reflexpr(this::function)` – the enclosing function.
- Helpful in several use cases
  - Implementation of logging
  - Implementation of cross-cutting aspects
  - ...