Document Number: P0256R0
Title: C++ Reflection Light
Audience: SG7 - Reflection
Author: Cleiton Santoia Silva

0.1 C++ Reflection Light [intro]

This is an attempt to summarize in few pages some important topics about the main SG7 proposals so far. This is also made by the author of one of the proposals, so it's biased, be warned !

0.2 Motivation for this paper [motivation]

— Too big reflection proposals, that happens because reflection imposes many cases and many more tricky corner cases;

— I want to point out the lack of few important foundations (not elaborate, just point) :

1. User defined attributes, so user could attach any literal type instance as attribute.
2. Constexpr strings or template string literal parameters([Tom14, N4121] or [Smi13, N3599]);
3. Typelists as language construct, this probably would help to make almost everything as type_-traits;

— C++ can do static reflection which is not so extensive used today as dynamic reflection, the current expectation are influenced by those dynamic frameworks, to see what can be done without a dynamic runtime map and without 'variant' nor 'object' parent of all see JSON serializer samples in [MCag16, P0194R0] and in [SA16, P0255R0] ;

0.3 Easiest case [easy]

Get the pointers of data members of T;

— [ATzag15, N4428] way:

```
int size = std::class_traits<T>::class_members::size;
// apply template meta to recurrently pick each I, from 0 to size - 1
auto pointer = std::class_traits<C>::class_members::get<I>::pointer;
// after that you must sort out what is data member and what is not thru the type of the pointer
```

— [MCag16, P0194R0] way:

```
typedef reflexpr(T) meta_T;
typedef std::meta::get_all_data_members_t<meta_T> meta_DMs;
int size = std::meta::get_size_v<meta_DMs>; // get number of data members
// apply template meta to recurrently pick each I, from 0 to size - 1
typedef std::meta::get_element_t<meta_DMs, I> meta_F;
auto pointer = std::meta::get_pointer_v<meta_F>;
```

— [SA16, P0255R0] way:

```
auto pointers = std::make_tuple( typedef< T, is_member_object_pointer >... );
```

ok..... that was a little biased sample, but a fairly common one;

0.4 Comparative [comparative]

— **Type traits x operator**, type traits are easier, but only operators can be used on namespaces;

— **Template expansion x Fill a structure**, apparently is easier to directly build your own structure upon a template expansion (hail tuples, enjoy template folding), than to be forced to use the standard structure and then convert it to your structure, however there are cases impossible to handle without some structure, like constructors or namespaces.

— **Pre-defined x query**, one may pre-define in the framework what you will get as `reflect(T).member_functions()` , and fill with a lot of functions, one for member objects, one for member functions, etc... but I think that is better to be able to create your own criteria and retrieve only what is needed (via concepts).

0.5 Other Issues [Other]

— Resurrect typeid(T), the intention behind type_info was run-time as in [ac12, N3437], however it's difficult because the design of type_info must change a lot, must become a constexpr and probably copyable also;

— The [ATzag15, N4428] and [MCag16, P0194R0] have reference implementations, [SA16, P0255R0] still don't.

0.6 Private Ryan [Private]

Can we rescue ryan member ?

```
struct X {
private:
 int ryan;
};
```

We don't know exactly what to do here, but we know from feedback in Urbana Meeting, what not to do ! We should not be intrusive in the class (change somehow the class to grant access), and also we don't want accidental access to private parts, somehow a reflection_unsafe() operator or option must be implemented.

0.7 Frankstein [frank]

— Better use traits whenever they fit, single return and typed argument;

— Operator to handle "not typed" arguments;

— Return a unnamed structure the cases that only the type or a const is not possible (namespaces, constructors, bit fields, modules, reference members);

0.8 SQL analogy [SQL]

One can make an analogy with SQL to facilitate (or not)

```
SELECT name, pointer, attribute A
FROM   type X
WHERE  member is object pointer and has attribute A
```

All tree groups of arguments may be passed to reflection operator, the fields from each member you want, the type that will be reflected and for what members you want as below.

```
reflect(X, {name, pointer, attribute_of_type<A>}) requires
is_member_object_pointer && has_attribute<A>;
```

0.9 Normal Cases [Normal]

1. Allow all types as parameters a.k.a. `reflect(int);`

2. Allow not typed parameters a.k.a. `reflect(namespace)`;
3. Allow to choose what to get
   a) `reflect(T).member_functions()`, `reflect(T).member_data()`
   b) I prefer `typedef<T, is_member_function_pointer>`
   c) What if i want just the int members, how can i use my own predicate ?
4. Allow token to type a.k.a. `class.forName("my_class")`;
5. Allow intended access to private parts, but not accidentally.

0.10 Difficult Case [Difficult]

1. Template member instantiations, two consecutive calls on reflect(T) may bring different results due to template instantiations.

0.11 Few acks [ack.ack]

Thanks to these particular individuals: Matus Choclik and Ricardo Fabiano de Andrade.

Bibliography

[ac12] Axel Naumann (Axel.Naumann at cern.ch). N3437 - type name strings for c++. Technical report, Programming Language C++, 2012.

[ATzag15] Christian Kaeser <christiankaeser87 at gmail.com> Andrew Tomazos <zos at google.com>. N4428 - type property queries (rev 4). Technical report, SG7 - Reflection Study Group, 2015.

[MCag16] Axel Naumann (Axel.Naumann at cern.ch) Matus Chochlik(chochlik at gmail.com). P0194r0 - static reflection (revision 4). Technical report, Programming Language C++, 2016.

[SA16] Cleiton Santoia Silva and Daniel Auresco. P0255r0 - c++ static reflection via template pack expansion. Technical report, C++ ISO Standard, 2016.

[Smi13] Richard Smith. N3599 - literal operator templates for strings. Technical report, Programming Language C++, 2013.

[Tom14] Andrew Tomazos. N4121 - compile-time string: std::string_literal<n>. Technical report, Programming Language C++, 2014.