Doc. no.: P0954r0.
Date: 2018-02-11
Programming Language C++
Audience: EWG
Reply to: Bjarne Stroustrup (bs@ms.com)

What do we want to do with reflection?

Bjarne Stroustrup

Introduction

We now again have several proposals for reflection, several proposals for metaprogramming and suggestions to merge the two areas. Outside the committee there is even more activity. This is good and reasonable, but how do we choose? The activity is a sign of great need. How do we get something that's good enough, soon enough? I fear delays from a quest for perfection. I fear mission creep.

I suggest we make a collection of key examples, so that we can compare the elegance of solutions for these, as well as their associated costs.

My examples

Let me start by offering four examples, that I think would serve most of my needs. I don't imagine that those are all we collectively want – the various proposals give more examples – but they may be a good start of a collection. I hope proposers will use that collection as part as their proposals, so that we can compare apples to apples. I hope that people will add their own examples to this collection.

To avoid pre-judging the solutions, my examples are deliberately very simple (naïve?).

If I had just what is mentioned here, many uses of C++ would be revolutionized or dramatically simplified. I'd rather not wait 10 years for this revolution.

File and Line

There are not many reasons left for using macros, but if we want to do logging, identification of the location of an assert, or in other ways would like to identify a location in a file, we need to use **__FILE__** and **__LINE__**. How about a pair of intrinsic functions?

```
Logger("this is a log entry from file {1} line {2}",std::file(),std::line());
```

Why would this be better than using the macros? Because "intrinsic functions":

- eliminate yet another reason for using macros and thereby decrease the temptation to users for defining their own.
- are easier to represent in a high-level program representation, improving the chance for better tool support.
- allows computer-generated code (e.g., generated from reflection) not to be preprocessed.

I/O functions

Writing I/O functions, serializers, and the like is essential for many applications, and a pain. I'd like to write something like

```
Stream_io<X>;
```

And have the output be something like:

```
istream& operator>>(istream& is, X& x) { /* read members is into x using >> */ return is; }
ostream& operator<<(ostream& is, X& x) { /* write members x to os using << */ return is; }
```

Basically, we have to see ("inspect") if **X** already has **>>** and **<<** and if not iterate over the members to generate the bodies of the **>>** and **<<** functions. I'd like one version of **Stream_io** for use inside **T** that could access private members of **X** and generate member functions and another version of **Stream_io** for use outside **X** that could not access private members.

The point about access to private members is debatable. We should have that debate: Should it be possible to introspect private data without touching the source of a class?

Ideally, the output would be inserted in the internal representation of the program where the request was encountered. If a compiler's internal representation can't reasonably be mutated, the output would have to be placed elsewhere and then linked in.

All proposals can do that, but how simply, elegantly, and cheaply can they do it?

Object maps

Dynamic reflection relies on data structures representing types. One obvious use of static reflection is to generate such structures, often to match the requirements of an existing dynamic reflection system (e.g., tool generated or relying on code annotations).

I'd like to write something like this

```
Map<X>;
```

Assume that **X** is

```
struct X { int i, string s; };
```

The output should be something like

```
Map_of_maps["X"] = {
        {representation_of("int"), offset_of("i")},
        {representation_of("S"), offset_of("s")}
};
```

Like for serialization, the basic solution is a loop over a class.

The **representation_of("S")** is where we must be able to represent types at run-time. The other use cases in this note require only that we have a standard way of accessing some or all that the compiler

knows about a type. That may be sufficient for initial use. If we can't represent types at run time, the example becomes

Map_of_maps["X"] = { {offset_of("i")}, {offset_of("s")} };

To use that, we would have to use static reflection plus some ad-hockery. If we do want to represent types at run-time, we'd need a standard representation. In that case, I suggest people have a look at [5]. I see the representation of types (and code) and access to it the subject most likely to cause mission creep and/or compile-time overhead (e.g., from a large internal representation) .

Function maps

Object maps give us the ability to write code that manipulate data structures without compiling those structures into a program, but how can we invoke operations on those? Some operations will be generic – basically interpreting the data structure and knowing what to do with a set of compiled-in basic types. Similarly, we sometimes want to represent an operation name as a text string (e.g., typed into the program by a user) and want it invoked. For that we need a map from strings to function pointers.

I'd like to write something like this

Function_map<X>;

Assume **X** is

struct X { void f(); string g(string); };

The output should be something like

Map_of_function_maps["X"] = { {"f",&X::f}, {"g",&X::g} };

Like for serialization, the basic solution is a loop over a class. Note that I left the type of those functions out of the map; that's just because I don't have a favored solution to how to represent those. The function map should produce something that is easily usable for a call to **std::INVOKE**.

I do not know how I'd prefer to express a wish for generating mappings for free-standing functions. I could write

int f(string);

Map_function<f>;

to have **f** added to some map of freestanding functions. The snag here is to get all mapped function into a useable data structure. That might imply run-time initialization or linker support.

Stringification

I don't personally feel the need to generate string versions of enumerator names (or other program entries, but I know that many do, so I list it here. Given

Enum E { red, blue green };

We should be able to say something like

Stringify<E>;

and get something like

string E_strings[] = { “red”, “blue”, “green” };

Questions about solutions

- How easy is a solution to use? The ideal answer is “trivially.”
- How much compile-time overhead does a solution impose? The ideal answer is “next to nothing.”
- How much run-time startup time does a solution impose? The ideal answer is “none.”
- How much run-time overhead is required to use an entry from a map? The ideal answer is “as much as an **unordered_map** lookup plus the inevitable indirection to access an element (data or function through function pointer).”
- How much data is taken up at runtime? The ideal answer is “just the **unordered_map**s from strings to necessary information for the classes and functions requested by the user.”
- Is there a way of representing types at run-tine? If so, is it general? And is it a potential standard?

Consider some types of arguments (obviously stereotyped) sometimes heard:

- *Bad*: My proposal is better than your proposal because your proposal cannot do X and mine can do X and B!
- *Good*: My proposal is better than yours because if can do just Z and does it simply, cheaply, and very well
- *Best*: This proposal is great because it can do just X, Y, and Z and does them simply, cheaply, and very well. X, Y, and Z is minimal and sufficient for most real-world users. If experience shows that we also need A and B, then here is how we might do that.

References

There are many papers on reflection and related metaprogramming in C++. This is not meant to be an exhaustive list.

1. Herb Sutter: Metaclasses: Generative C++. P0707R2.
2. Daveed Vandevoorde: Reflect Through Values Instead of Types. P0598r0.
3. Daveed Vandevoorde and Louis Dionne: Exploring the design space of metaprogramming and reflection. P0633R0.
4. Matúš Chochlík, Axel Naumann, David Sankel: Static reflection of functions. P0670R1.
5. [Gabriel Dos Reis, and Bjarne Stroustrup: A Principled, Complete, and Efficient Representation of C++. Proc. Joint Conference of ASCM 2009 and MACIS 2009. COE Lecture Note Vol. 22, pp. 407-421; December 2009.

Reflection talk