Generative Extensions for Reflection

P3157R0

Andrei Alexandrescu
NVIDIA
andrei@nvidia.com

Bryce Lelbach
NVIDIA
blelbach@nvidia.com

Michael Garland
NVIDIA
mgarland@nvidia.com

I. Introduction

Since the recent implementation of the reflection facilities proposed in P2996R1 [1], we explored how the proposal would help fundamental challenges we are facing today. We believe reflection has great potential to solve important problems that we and the C++ community at large both face, and that a few specific enhancements on top of P2996R1 would help realize that potential. Based on our initial experience, this document sketches a few specific extensions built on the foundation of P2996R1 that we think will help C++ reflection have a stronger and more timely positive impact on the state of affairs in the C++ language. We anticipate that C++ reflection, if sufficiently powerful, could dramatically reduce costs of developing and maintaining C++ code while at the same time improving code size, readability, compilation time, and execution speed.

C++ code from a variety of domains naturally leads to boilerplate. Proxy classes are commonplace, whether to interface different codebases or as an organic part of design. The guideline "prefer composition over inheritance" [2] leads to many forward-to-member functions. Other useful design patterns such as Visitor, Observer, Decorator, Adapter [3], and Null Object [4] require de facto maintenance of parallel class hierarchies and/or parallel function declarations (plus in many cases mechanical definitions). Foreign language/API interfaces typically consist of large swaths of code following repetitive patterns. Implementing high-performance parallel algorithms typically requires specialized patterns for a variety of size, stride, and type combinations. All of these instances feature enough irregularities and variations to make existing template metaprogramming techniques difficult to deploy, and if deployed, difficult to understand and maintain.

The following sections define use cases of reflection metaprogramming that guide the upcoming design and implementation. We consider strong use cases essential to setting goals for reflective metaprogramming. To be compelling, use cases must demonstrate meaningful, desirable functionality (albeit in a simplistic, proof-of-concept style) that is impossible or prohibitively difficult within the current C++.

II. Proxy classes and instrumented classes

Consider the task of defining interface classes for API integration, including foreign language bindings. Such a task involves creating classes with member functions that perform the same basic duties—such as validating arguments, converting data formats, and adjusting reference and pointer notations—before calling similarly named functions in another class, sometimes followed by analogous postprocessing of the results.

Such use cases generalize to creating *instrumented classes:* developing direct substitutes for existing classes while embedding specific hooks (such as tracing, logging, counters, argument verification, result validation, and naming convention changes) into some or all member functions. Successfully reflecting on and reconstructing a class in this manner serves as a critical test of a language's reflective metaprogramming capabilities, similar to how the identity function evaluates a language's functional programming features.

As a simple example, consider the Null Object design pattern, a safe alternative to the dreaded null pointer. Given an interface `T` that defines several pure virtual member functions, `null_object<T>` would yield an implementation of `T` that defines all of its virtuals to either throw an exception or return a default-constructed value of their result type. Defining a null object for a given interface is tediously simple, yet maintaining it is an exercise in frustration and a source of aggravation. Reflection should allow defining `null_object` for any type and behavior with ease.

A more involved example would be defining a class such as `instrumented_vector<T, A>` that wraps an `std::vector<T, A>` and adds instrumentation (e.g., bounds checking) to some or all of its methods. The key here is to make `instrumented_vector` a drop-in replacement for `std::vector` without incurring the cost of copying all of its member function declarations. Needless to say, defining such a proxy class by hand is quite discouraging, and reflective metaprogramming should offer a complete and flexible solution.

The ability to create proxy classes would also put to rest the unpleasantness of following the adage "prefer composition over inheritance" in C++. As mentioned, in many composition situations, numerous forward-to-member functions must be written and maintained; automating these stubs would make it much easier to follow the guideline therefore improving code quality without adding to its bulk. Making valuable programming idioms more accessible has repeatedly proven to be a wise investment.

An important part of defining instrumented classes is querying all members of an existing class (static and nonstatic data member, regular and special member functions, enum declarations, friend declarations…) and generating similar definitions within the context of a new class definition. The `define_class` primitive in P2996R1 is the fundamental mechanism for implementing a proxy class, and although it currently does not support adding member functions, it alludes to such a possibility in section 4.4.12: "For now, only non-static data member reflections are supported (via `nsdm_description`) but the API takes in a range of `info` anticipating expanding this in the near future." We believe the ability to define full-fledged classes is a quintessential, defining feature of a reflective metaprogramming feature for C++. Here are a few key components needed:

- Signatures of all functions must be accessible for introspection, and primitives for accessing full information of a function's signature must be defined.
- Synthesis of function signatures must be possible, e.g. a library may need to build a signature from scratch, or from a similar signature (e.g., create a new signature from a given signature by adding or removing an attribute).
- There must be an ability to attach code to the reflection of a function signature; for example, a library may want to define a proxy class that inserts logging for each function's arguments and result. The most fit candidate for attaching such functionality to reflection is a generic function literal that is a friend of the generated class.
- Finally, `define_class` would accept synthesized member functions in addition to (and in a manner similar to) `nsdm_description`. Implementing member function synthesis should be feasible following a design similar to `nsdm_description`—a function `memfun_description` would take an `std::meta::info` (either synthesized or coming from the introspection of another member function) a `memfun_options` object that has, among other members, a lambda function to serve as the body of the budding member function. The result of the call to `memfun_description` would be passed to `define_class`.

One aspect of function synthesis is *code cloning*—the ability to compile a reflected function template under different constraints (e.g. different concepts and attributes). As an example, this aspect is important to CUDA C++ libraries that need to add **`__device__`** attributes to all methods of replicas of standard types—such as `std::pair`, `std::tuple`, and `std::optional`—and subsequently compile the resulting code for use on the device. The alternative—copying and pasting code with minute changes—is a proverbially bad practice.

III. Closure reflection

Libraries that perform any sort of serialization across address space boundaries (such as saving/loading data to/from files, transporting it over a network, copying to other processes, or marshaling to a GPU with a distinct address space) must be mindful about pointers; they should be able to perform transitive copying, address fixup, or reject the serialization attempt outright.

Introspection as proposed in P2996R1 does offer the ability to reflect on the data members of a class, but not on the captures of a lambda expression. For completeness, full introspection should be available for closure objects. The C++ standard does not prescribe closures to be class types in order to grant leeway to the implementations. This means an introspection feature would need to explicitly guarantee that closure introspection is available (in addition to class introspection).

We note that defining this feature is mainly a matter of wording; implementation is likely very simple because implementation-wise closures are essentially class types that define operator() and a few special functions. In fact, the current implementation available via Compiler Explorer treats lambdas and classes in the same manner.

IV. Complete and accurate reflection of representation

The reflection facility should define a (meta)function `representation_of<T>` to take any type `T` and yield a plain **`struct`** that mimics the exact data layout of `T` (in such a way that a reference to `T` can be cast to a reference to `representation_of<T>` and have addresses of all members in correspondence). For example:

```
class X {
    char a;
    alignas(8) int b;
    virtual ~X();
public:
    double c;
};

// representation_of<X>, generated via reflection, looks like this:
template<> struct representation_of<X> {
    void* __vptr0;
    char a;
    alignas(8) int b;
    double c;
};
```

Such representation types are crucial for low-level manipulation of typed data, and an enabler of a variety of other data manipulation techniques such as hashing, formatting, parsing, serialization, marshaling, and networking.

Currently defining `representation_of` is not possible; the array returned by `nonstatic_data_members_of` does not include the compiler-introduced members. (Thankfully alignment can be both retrieved with `alignment_of` and set with `nsdm_options_t::alignment`.) We estimate this feature would be easy to implement but must be present in the wording and in the compiler-provided implementation.

V. Embedded Domain Specific Languages

Python's many frameworks (e.g., PyTorch and TensorFlow, among many others) have demonstrated the importance of Embedded Domain Specific Languages (EDSLs) in today's world, especially for AI applications. A reflective metaprogramming facility for C++ is expected to make it possible to express EDSLs much better than C++ currently allows.

EDSL-related features would place emphasis on the *generative* aspect of reflective metaprogramming. For example, an aspirational EDSL way of doing things for GPU-accelerated code could take in an algorithm written concisely in a high-level array language and generate during compilation specialized CUDA C++ code implementing the algorithm with the same efficiency as if this low-level code were written by hand.

The utility of EDSLs is, of course, much broader. An EDSL essentially allows the programmer to author a high-level expressive specification adapted to the problem domain (function differentiation, relational databases, networking protocols, regular expressions, EBNF grammars, document formatting…), to then generate C++ code from it. The approach is advantageous if writing the same C++ code by hand would be a much more costly proposition. Domain-specific libraries can provide the desired level of abstraction, but struggle to optimize execution in ways that cross abstraction boundaries. EDSL-style approaches can provide this missing capability.

The *amplification* aspect is an important desired outcome: the ultimate goal of a reflection facility is to allow new code to build on existing code in a combinatorial manner, to the effect of automating tedious and repetitive aspects of the coding process. To wit, the examples in P2996R1 and those shown above invariably would take more code to implement without reflection. Allowing generation of meaningful code from specifications encoded in EDSLs would be the ultimate goal of a reflection engine.

EDSL processors (toolchains that act as embedded interpreters or compilers) play a pivotal role in the EDSL ecosystem. Though an EDSL processor itself can be seen as an ordinary EDSL that takes in a grammar specification and produces a language translator, once the EDSL toolchain is available it can be used to process any other EDSL, either during compilation or at runtime. Therefore, an EDSL toolchain defined as an EDSL is a foundational "seed EDSL" that we consider a crucial milestone of a reflection metaprogramming engine.

For such a task to be feasible, the *synthesis* aspect of the proposal is important—building on the `memfun_description` function discussed above, we envision adding primitives that synthesize types, functions, and templates in addition to the `define_class`/`nsdm_description` facility that allows definition of classes with direct data members.

References

[1] W. Childers, P. Dimov, B. Revzin, A. Sutton, F. Vali, and D. Vandevoorde, "Reflection for C++26," https://open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2996r1.html, Dec 2023.

[2] "Composition over inheritance - Wikipedia," https://en.wikipedia.org/wiki/Composition_over_inheritance, (Accessed on 02/15/2024).

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995.

[4] K. Henney, "Null object," in *Proceedings of the 7th European Conference on Pattern Languages of Programs (EuroPLoP'2002)*. Citeseer, 2002.