Document: P2062R0
Revises: (original)
Date: 01-11-2020
Audience: SG7
Authors: Wyatt Childers (wchilders@lock3software.com)
Andrew Sutton (asutton@lock3software.com)
Faisal Vali (faisalv@yahoo.com)
Daveed Vandevoorde (daveed@edg.com)

The Circle Meta-model

Introduction

During the November 2019 meeting in Belfast, some of the SG7 participants enthusiastically mentioned Circle[1] as providing a more intuitive compile-time programming model and suggested that SG7 investigate overhauling the de-facto SG7 approach (P1240+P1733[2]) to follow Circle's general approach (to reflection and metaprogramming).

This paper describes a framework for understanding metaprogramming systems and provides a high-level overview of some of Circle's main characteristics, contrasting them to P1240's approach augmented with an injection mechanism along the lines of P1717.

While we appreciate some of Circle's powerful capabilities, we also raise some concerns with its underlying model and provide arguments in support of P1240's choices as being a more suitable fit for C++'s evolution.

The Dimensions of Reflective Programming

In P0633, we identified three "dimensions" of compile-time reflective metaprogramming:

1. Control:
   How are compile-time computations effected/interpreted? What are metaprograms?
2. Reflection:
   How are source constructs are made available as data for use in metaprograms?
3. Synthesis:
   How can "code" be generated from a programmatic representation?

[1] https://www.circle-lang.org/ Circle is an impressive project: Sean Baxter developed a brand new C++17-like front end on top of LLVM, incorporating a variety of new compile-time capabilities that align closely with SG7's goals. For Sean's motivations, see https://github.com/seanbaxter/circle/blob/master/examples/README.md#why-i-wrote-circle.

[2] P1733 suggests building a type hierarchy on top of the monotype representation suggested by P1240. That was in principle agreed on by SG7 and work is ongoing to make that possible.

These dimensions can help us understand metaprogramming systems by decomposing them into largely independent core components.

Current proposals for metaprogramming are already “layered” in these terms. We have a number of proposals to extend constant expression evaluation (control), P1240 defines a comprehensive system for compile-time reflection and reification (reflection), and P1717 proposes a template-like mechanism for code injections (synthesis)[3].

The adoption of Circle would represent a significant addition to the language and its design incorporates specific choices for each of the three questions above. We think that examining those choices independently is useful to shed light on whether those choices are desirable for C++.

Reflection

P1240’s approach to reflection introduces a single keyword (`reflexpr`) to create an expression reflecting a source construct, and then transform those values using (`consteval`) functions. For example, to get a representation of the members of a class `X`, you could use the expression `members_of(reflexpr(X))`, which produces a vector of reflections for the members of `X`.

Circle approaches this differently.

The first observation is that Circle has carved out a separate namespace for keywords by introducing the character `@` to the basic source character set: This allows the introduction of dozens of new keywords such as `@meta`.

The second observation is that Circle does not actually provide true reflection in the sense of introducing one or more types that correspond to reflections. Instead, it combines reflection and reification into new operators called “introspection keywords”. For example, `@member_type(X, 3)` is a *type-specifier* that produces the type for the fourth nonstatic data member of `X`. Doing the same in the P1240 model requires both reflection and reification:

```
typename(members_of(reflexpr(X), is_data_member)[3]).
```

The latter is more verbose, but also more composable.

There are currently about a dozen documented introspection keywords in Circle, which fall in three categories:

1. Introspecting nonstatic data members (`@member_count`, `@member_name`, `@member_ptr`, `@member_value`, `@member_type`)

[3] This injection mechanism was also presented at CppCon 2019. See https://www.youtube.com/watch?v=kjQXhuPX-Ac.

2. Introspecting enumerators
   (`@enum_count`, `@enum_name`, `@enum_value`, `@enum_type`)
3. Conversions between string values (e.g., `"int*"`) and types (e.g., `int*`)
   (`@type_string`, `@type_id`)

In its current form, this is considerably more limited than the nearly-exhaustive API proposed in P1240, but plenty of additional introspection keywords could be added to the Circle set. The fundamental difference is that Circle does not traffic in a reflected representation; there is no first class reflection value.

Synthesis

P1240 currently provides a handful of reification primitives to turn reflections back into source constructs, and otherwise depends on the template instantiation mechanism (including *expansion statements*; see P1306) to compose reifications into more complex constructs. Additional work is being done to explore "code injection". Previous presentations to SG7 suggested both string injection (compose a compile-time string value and inject it in the source code) and token-sequence injection (allow injecting a sequence of tokens containing reification primitives in various contexts), but those options were voted against. Andrew Sutton and his colleagues at Lock3 Software are working on a higher-level "semantic injection" mechanism that builds on the P1240 ideas. This approach is documented in P1717.

Circle essentially uses token-sequence injection as previously presented to SG7 (but rejected at the time). There is however an important difference in composability. To explain this, let's look at a bit of Circle injection code:

```
@meta for(size_t i = 0; i < @member_count(arg_t); ++i) {
  push(@member_value(object, i));
}
```

Here the `@meta for` statement is executed at compile time and the statement it controls is "injected" in its surroundings. "Surroundings" for Circle-style injection means "floating up past `@meta` scopes". For example:

```
void f(X &object) {
  @meta if (sizeof(X)<100) {
    @meta for (size_t i = 0; i < @member_count(arg_t); ++i) {
      push(@member_value(object, i));
    }
  }
}
```

Here the token sequence "`push(@member_value(object, i));`" is injected (or, "floats up") zero or more times into the outermost block of function `f`, with `@member_value(object, i)` replaced by an expression that produces an lvalue for the (i+1)st member of `object`.

What if we wanted to create an abstraction for this injection? A Circle injection cannot be moved to a function because the injection would now inject in that function instead of in its caller, but it does introduce its own kind of hygienic macros (defined with `@macro`) to work around that. So, we could write:

```
@macro m(X &object) {
  @meta if (sizeof(X)<100) {
    @meta for(size_t i = 0; i < @member_count(arg_t); ++i) {
      push(@member_value(object, i));
    }
  }
}
void f(X &object) {
  m(object);
}
```

Note, however, that that is fundamentally different from the other approaches contemplated in SG7: The macro invocation doesn't just re-parse the injected code, but also the control code. In contrast, all the consteval-based injection methods that are being considered inject either in the last consteval-block that was entered (in temporal terms), or in a specific context that is specified as part of the injection primitive. The example above might be written in P1717 as follows:

```
#include <meta>
using std::meta::info;
consteval m(info object) {
  if (byte_size_of(object)<100) {
    for (member : members_of(object, is_data_member)) {
      -> fragment {
        push(exprid(object).exprid(member));
      }
    }
  }
}
void f(X &object) {
  consteval { m(reflexpr(object)); }
}
```

Unlike in the Circle macro-based approach, the function `m` is only parsed once. In fact, even the fragment itself is also parsed just once, but every time the injection operator (prefix `->`) is *evaluated*, the fragment is queued up to be expanded (an AST expansion in the current implementation). This expansion occurs in the context just following the last `consteval {...}` block that was entered.

Control

The “control” dimension of reflective metaprogramming is where Circle most fundamentally differs from the `consteval`-based metaprogramming models SG7 has looked at before.
The Circle compiler integrates an interpreter that is *bit-accurate* with respect to the C++ ABI with which the Circle compiler is implemented (the “host environment”). This interpreter will evaluate expressions, including calls to any function whose definition is available (i.e., not just constexpr functions).

Because it is bit-accurate, the Circle interpreter can also call functions compiled natively in the host environment. That in turn, for example, makes the <iostream> implementation of the host environment available to meta-code; compiling the following bit of code:

```
#include <iostream>
void f() {
  @meta std::cout << “Hello, world!\n”;
}
```

will output “Hello, world!” to the console in which the compiler is invoked, and produce object code for

```
#include <iostream>
void f() {
}
```

The Circle compiler has options to specify additional shared libraries that can be loaded and invoked from meta code. So, while the Circle interpreter itself is not particularly fast, it is possible to separately compile code into a shared library for the host environment, and other translation units can then invoke that (fast) shared library during compile-time evaluation[4]. The compiler mightlink calls to shared libraries for use at compile time using the ABI (e.g., calling convention) of the platform that the compiler is executing on, while generating an executable image for the target consistent with the target's ABI (and *its* calling convention)

[4] This adds an interesting question to the C++ program model. Is a shared library used only for compile-time evaluation considered to be part of a program? With C++ constant-evaluation, the answer is always “yes”: A program can only execute the functions it declares. With Circle, the answer could conceivably be “no”.

Currently, the publically-available Circle compiler has a “target environment” that is mostly identical to its host environment. That allows the scheme described above to work seamlessly. For a similar scheme to work in a full cross-compiler[5], however, new challenges arise. Consider:

```
template<size_t N> struct X;
template<> struct X<sizeof(vector<int>)> { ... };
```

In C++, currently, that specialization is for the size of a `vector<int>` in the target environment. In the host environment, that may be a different size.

Similarly, if we were compiling code for some big-endian target using a compiler hosted on a little-endian machine, the compile-time code would execute under the host-platform (i.e., little-endian), thus the same expression could give us wildly different results at compile time vs. run time:

```
int f() { int num = 1; return *(unsigned char*)&num; }

assert(f() == 0);  // Fails at run-time (big-endian).
@meta assert(f() == 0);  // Okay at compile-time (little-endian).
```

A Circle-like compiler could compile the code twice, but in that case the programmer must be careful to always keep in mind the dual nature of all types (e.g., `vector<int>` could have two distinct sizes). Alternatively, the compiler could integrate a low-level virtual machine emulating the target architecture, but that is likely not practical (e.g., the emulated architecture might not itself have `std::cout`) and discards the performance advantage of native host-side execution. It would also require the development of a virtual machine for every target architecture, which is likely economically prohibitive.

In contrast, the constexpr evaluation model has always been in terms of the target environment. The host environment is simply not exposed to the formal source code[6]. A constexpr evaluator need not be (and typically *isn’t*) bit accurate, but that also means it *can* be significantly faster than a bit-accurate interpreter. Programmers in that model do not have to concern themselves with every type potentially having dual representations.

Opinion

Circle is an awesome project, all the more so because it is also a C++17 implementation developed “from scratch” by a single dedicated programmer in just a few years.

[5] By “full cross compilation”, we mean that the host and target machines are completely disjoint. The Circle implementation does have the ability to generate code for CUDA devices, which involves a kind of cross compilation. However, compile-time native code invocation still requires native host-side support.
[6] The host environment *is* of importance to the *physical* source code since character encodings matter. Fortunately, those encodings are largely independent from specific environments. See also P1953.

However, we do not believe its metaprogramming model is the right direction for C++'s future. We raise the following concerns:

- Executing code in the host environment considerably complicates cross compilation (which is a *very common* scenario[7]).
- The ability (and potential *need*) to call into shared libraries from the compiler raises the kinds of security concerns that led SG7 to discard `std::embed` (P1040).
- Host-side native shared libraries also make it more difficult to have reproducible builds and constrain the tool ecosystem in general (in practice, every significant tool would want to be implemented using the same ABI/compiler as the compiler itself).
- The injection mechanism is overly dependent on (hygienic) macro expansions, which limits scalability.
- The inability to separate reflection and reification limits the flexibility and scalability of the introspective facilities.

During the discussions in Belfast, and in private discussions afterward, one argument that has been made is that having the ability to use the <iostream> interface for compile-time I/O is desirable. We believe that that is achievable within the framework we propose[8], but also that it is a sizeable project. Meanwhile, we can develop a smaller useful interface on top of which "compile-time streams" can be developed.

In conclusion, we argue that SG7 should continue with the P1240+P1733 model for reflective metaprogramming, with the addition of a suitable injection model such as the fragment-based solution described in P1717 (a simpler token-based injection system has advantages, too, but it makes certain kinds of tools difficult or impossible).

[7] We don't have specific numbers for the fraction of C++ projects that rely on cross compilation, but it includes most of mobile application development, embedded application development, and even some server applications.
[8] Not through `std::cout`, `std::cin` and the like, which remain handles to the target environment, but through some other "host streams" that implement the `<iostream>` interfaces.