constexpr C++ is not constexpr C

Document #:	P1447R0, ISO/IEC JTC1 SC22 WG21
Date:	2019-01-21
Project:	Programming Language C++
	SG7, Evolution, Library Evolution

Reply-to:	Matúš Chochlík
	<chochlik@gmail.com>
	Axel Naumann
	<axel@cern.ch>
	David Sankel
	<dsankel@bloomberg.net>

Contents

1 Introduction

Reflection is moving towards `constexpr` value-based notation, becoming the first of its kind in the C++ standard library. We as a community must decide if we want `constexpr` value-based libraries to follow best-practice, C++ design patterns in spite of compile-time performance concerns. The authors believe that `constexpr` libraries designed as a collection of C-style free functions without strong types would be a disservice to the community, and be at variance with C++'s core values. We provide explanations and numbers to support this position.

2 Members or Free Functions? Syntax and Type Safety

The discussion boils down to the following pseudo-Tony-Table:

Monotype/Free-function style [P1240R0]	Rich types/OO-style [P0953R2]

`meta::info str_m = reflexpr(string);`
`vector<meta::info> mems`
`  = reflect::get_members(str_m, is_type);`
`meta::info first = mems[0];`
`string name = meta::name_of(first);`
	`reflect::Record str_m = reflexpr(string);`
	`vector<reflect::RecordMember> mems`
	`  = str_m.get_member_types();`
	`reflect::RecordMember first = mems[0];`
	`string name = first.name();`

The object-oriented, type-safe, programming style remains a software engineering best practice to this day. Many of the newest, fanciest libraries employ these patterns as they strive for for clean, conceptually-simple value semantics. Thanks to C++, software engineers have access to these quality libraries.

Back in the 90s, compilers were incredibly SLOW compared to C. Even today, compiling C++ is still slower than compiling C. Yet, C++ survives and thrives. Why should we give up all the benefits of OO design for `constexpr`-based libraries? Why would different arguments hold for `constexpr`-based libraries than for regular ones?

The authors believe in the vendors' capability to accelerate the relatively new `constexpr` compilation model according to user demand. Our numbers below demonstrate that acceleration of `constexpr` compilation might not even be needed to take advantage OO design at compile time!

One of the primary motivations for replacing the Reflection TS's template meta-programming style [N4766] with `constexpr`-based interfaces is making compile time metaprogramming more accessible. Templates require operations to be placed to the left of their operand. Free-function-style `constexpr` programming keeps this "inverse Polish notation", while our preferred OO-style renders as readable, natural, code. The following table demonstrates this:

Reflection-TS [N4766]	Our preferred notation

`using X_m = get_scope_t<`
`  get_type_t<`
`    reflexpr(some_var)`
`    >`
`  >;`
	`auto X_m = reflexpr(some_var)`
	`  .get_type()`
	`  .get_scope();`

A major bonus of C++ is type safety: you can drive a car, not a cat, and the compiler will tell you. [P1240R0] instead suggests to tear down type safety from reflection. Instead a fundamental, opaque type is used for almost everything. Without type safety, [P1240R0] was even forced to introduce an "invalid" state of the untyped reflection "object". This is used, for instance, as the result of calling `get_members` on `reflexpr(int)`.

The object-oriented, type safe approach of [P0953R1] does not even offer this operation: `reflexpr(int)` yields a `Type` not a `Record`, and only the latter offers the interface `get_members()`. This is what we are used to (and love!) with C++.

To clarify, we could have written the first Tony Table as follows, with an implicit `using namespace std::meta` and `using namespace std::reflect`:

Free-function-style [P1240R0]

```
template <class T>
constexpr std::string getName() {
  info str_m = reflexpr(T);
  if (!is_valid(str_m))
    throw std::logic_error("T is invalid for reflection");
  vector<info> mems = get_members(str_m, is_type);
  if (mems.empty())
    throw std::logic_error("Zero members");
  info first = mems[0];
  constexpr std::string name = name_of(first);
  // handle invalid operation?
  return name;
}
```

Our preferred OO-style

```
template <class T>
constexpr std::string getName() {
  Class str_m = reflexpr(T);
  vector<RecordMember> mems = str_m.get_member_types();
  RecordMember first = mems[0];
  return first.name();
}
```

3 Performance

During the San Diego discussion of [P0953R1] and [P1240R0], the [P1240R0] authors raised concerns about the (compile-time) performance implications of [P0953R1]'s object-oriented notation. To measure the impact, we have conducted two benchmarks with the clang compiler. They provide an estimate of the upper and lower bounds of object-oriented `constexpr` notation's compile-time performance penalty.

3.1 Stateless

The small benchmark [constexpr-perf-stateless] generates constexpr evaluations of 10000 functions. Each provokes numerous instances of name lookup. The return values of all functions are "stateless" in that they do not require context; the compiler will immediately evaluate them to a constant value.

```
// p0953_head.cxx
#include <array>
enum Dummy{a, b, c};

struct Enumerator {
   // Using array because compile-time std::vector is not available
   constexpr std::array<char, 128> name() { return {}; }
   constexpr int value() { return 42; }
```

```
};
struct Enumeration {
   constexpr std::array<Enumerator, 3> enumerators() { return {}; }
};

constexpr Enumeration reflexpr() { return {}; }
    // Simulates 'reflexpr(Dummy)'
```

```
// p0953_one.cxx; cloned 10000 times, replacing '@' with 1, 2,...
template <class T>
constexpr std::array<char, 128> get_name_@(int v) {
   for (Enumerator e: reflexpr(/*T*/).enumerators()) {
      if (e.value() == v)
         return e.name();
   }
   return {};
}

constexpr auto eval@ = get_name_@<Dummy>(1);
```

This is compared to a [P1240R0]-style set of free functions, doing the same operations.

```
// p1240_head.cxx:
#include <array>
enum Dummy{a, b, c};

struct Info {};

constexpr std::array<Info,3> enumerators(Info) { return {}; }
constexpr int value(Info) { return 42; }
constexpr std::array<char, 128> name(Info) { return {}; }

constexpr Info reflexpr() { return {}; }
```

```
// p1240_one.cxx
template <class T>
constexpr std::array<char, 128> get_name_@(int v) {
   for (Info e: enumerators(reflexpr(/*T*/))) {
      if (value(e) == v)
      return name(e);
   }
   return {};
}

constexpr auto eval@ = get_name_@<Dummy>(1);
```

The source file generated by a concatenation of the respective `..._head.cxx` and 10000 clones of its `..._one.cxx`, where '@' is replaced by a running number, is then compiled on a MacBook 2014, 2.5 GHz Intel Core i7, with `-fsyntax-only` with

```
$ clang --version
Apple LLVM version 10.0.0 (clang-1000.11.45.5)
Target: x86_64-apple-darwin18.2.0
Thread model: posix
```

The resulting timings are

free-function, p1240	**object-oriented, p0953**
`real 0m2.969s`	`real 0m2.702s`
`user 0m2.858s`	`user 0m2.592s`
`sys 0m0.100s`	`sys 0m0.101s`

As one can see, the difference is within the range of "random" fluctuations.

3.2 Stateful

A more complete benchmark modifies recent clang from Wed, Jan 2, 2019, [clang-stateful] to enable a comparison where `constexpr`-state is provided through compiler intrinsics. This state is then passed along through either object-oriented of free-function coding syntax. The code to be benchmarked [constexpr-perf-stateful] exercises similar functionality and uses a similar setup to the code for the stateless benchmark.

free-function, p1240	**object-oriented, p0953**
`real 0m2,674s`	`real 0m4,020s`
`user 0m2,649s`	`user 0m3,987s`
`sys 0m0,024s`	`sys 0m0,032s`

The object-oriented approach is naive in that it uses a dedicated `this` pointer that needs to be evaluated to determine the (opaque) pointer value to the AST-node. On an Intel Core i5-2400 CPU @ 3.10GHz, this extra step costs about 50% in performance compared to the free function approach, where the (opaque) pointer value to the AST-node is the `Info` node itself. A smarter implementation of the object-oriented style could map the `this` pointer of the `constexpr` object to the internal AST-node for reflection objects, alleviating most of the overhead.

3.3 Conclusion

We can show that the lookup performance is approximately equal. On the other hand, the cost to pass the context / AST-node pointers depends on the implementation. Be believe that this can be optimized in several ways. Our benchmarks thus indicate that the minimum overhead of the object-oriented coding style is 0%, and the maximal overhead is 48%.

Note that this benchmark is purely synthetic; code that uses reflection operations in about 100% of its code will be extremely rare. The actual overhead in real code will thus be only a fraction of whatever overhead comes from object-oriented programming style.

4 Summary

Object-oriented design is simple to reason about and easy to write. It fits naturally into C++ and its focus on values, type-safety, and conceptual abstractions.

A collection of free functions suffers similar notational issues as template meta-programming does, where the operation is *followed* by the object it is operating on: `at(coll, idx)` instead of `coll.get(idx)`. In San Diego, many agree that object-oriented `constexpr`-libraries are preferred in principle.

This paper shows that the benefits of such an object-oriented `constexpr`-library is well worth the (small) cost, making [P0953R2] the preferred choice for a C++ standard reflection library.

5 References

[clang-stateful] Matúš Chochlík. 2019. Clang-modifications for a rudimentary, naive stateful reflection implementation.
https://github.com/matus-chochlik/clang/tree/reflexpr

[constexpr-perf-stateful] Matúš Chochlík. 2019. Stateful OO-`constexpr` benchmark.
https://github.com/matus-chochlik/mirror/tree/develop/benchmark/constexpr-perf

[constexpr-perf-stateless] Axel Naumann. 2019. Stateless OO-`constexpr` benchmark.
https://bitbucket.org/karies/constexpr-perf/src/master/

[N4766] 2018. Working Draft, C++ Extensions for Reflection.
https://wg21.link/n4766

[P0953R1] Matúš Chochlík, Axel Naumann, and David Sankel. 2018. `constexpr reflexpr`.
https://wg21.link/P0953R1

[P0953R2] Matúš Chochlík, Axel Naumann, and David Sankel. 2019. `constexpr reflexpr`.
https://wg21.link/P0953R2

[P1240R0] Andrew Sutton, Faisal Vali, and Daveed Vandevoorde. 2018. Scalable Reflection in C++.
https://wg21.link/P1240R0