Reflection-based lazy-evaluation

Document #: P2040R0
Date: 2020-01-13
Project: Programming Language C++
Audience: SG-7
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract

This paper aims to demonstrate how proposed reflection and code injection facilities can be used to replace many usages of the processors and provide simple lazy evaluation without introducing new constructs I believe the combination of what is presented here, reflection (including reflection on attributes), reification and injection would supersede all use of macro functions.

Disclaimer

This paper intends to present an idea that I had for a while so that it is on the table and can be considered along already proposed alternatives. However, it has not been implemented nor refined. It only aims to present a possible general direction. Rather than to propose new features, it aims to illustrate how the reflection and code injection proposals can be leveraged to express things that can currently not be conveniently expressed without the C preprocessor.

Building blocks

Reflection on expressions

This paper assumes the ability to reflect on arbitrary expressions as presented by [7]:

```
reflexpr(1+1);
reflexpr(std::cout);
reflexpr(f());
```

Expressions have a type, and so reflection of expressions have the same type, which might be queried with `typeof`. we can introduce a concept to describe expressions, which we could pass to `consteval` functions:

```
void f(meta::info expression) requires meta::is_expression;
//constrains on the type of the expression
void f(meta::info expression) requires meta::is_expression_of<int>;

f(reflexpr("not an int")); // ko
f(reflexpr(42)); // ok
```

We could further introduce a `expression` concept

```
template <meta::info I>
concept any_expression = meta::is_expression(I);
template <typename T, meta::info I>
concept expression = meta::is_expression(I) && convertible_to<typeof(I), T>;

consteval void f(std::expression<int> auto);
```

Although it is not clear to me how the facilities to constrained on values proposed by [7] and concepts would interact, we could then write the following:

```
consteval void f(std::expression<int> auto);
```

Code Injection

The ideas presented in this paper rely heavily on [P1717] [5]

The basic idea is that code fragments, whether expressions or statements can be injected in a given parent scope:

```
consteval void f() {
    -> { injected_call() };
}

void g() {
    f();
}
//becomes =>
void g() {
    injected_call();
}
```

`-> { injected_call(); }` represents an injected fragment (in this case a statement).

Please read [P1717] [5] for a better description.

Proposal: reflection/injection-based hygienic macros

By combining Reflection on expressions and code injection, it is possible to create a powerful mechanism to manipulate expressions and inject expressions and statements.

Once passed to a `consteval` function, reflections can either be evaluated if the reflected expression is a constant expression - but in the context of this paper, this has limited usefulness - Or injected in the scope in which the function is called.

Injecting the expression does not require it to be evaluated.

```
bool enable_logging = /*....*/;

void consteval log(expression<string_view> auto message) {
  -> {
        if(enable_logging) {
            std::cout << message << "\n";
        }
    };
}
```

The above code features a fragment injection statement as described in [6] and [8]. That fragment will be injected or expanded in the caller scope at the point of invocation.

```
int main() {
    log(reflexpr("Hello"));
}
```

Is equivalent to:

```
int main() {
   if(enable_logging) {
       std::cout << "Hello" << "\n";
   }
}
```

The important point to note is that the expression used as a parameter to the `code` function will never be evaluated when `enable_logging` is false.

Below, `expensive_computation` is never called.

```
std::string expensive_computation();
int main() {
    enable_logging = false;
    log(reflexpr(expensive_computation()));
}
```

As such, using reflection and code injection in that way can enable a form of lazy evaluation. However, `log` is expended more than it is called, which differs from [P0927][1].

It is likely both proposals would generate the same code if the callee can be inlined. Both proposal alleviate the use of a macro.

```
# define log(message) \\
    if(enable_logging) { std::cout << message << "\n" };
```

And, unlike processor macros, this reflection-based solution

- Obeys the name lookup rules of C++

- Obeys the scoping rules of C++ and can be declared inside a namespace or a class
- Is semantically checked at the point of declaration
- Can have typed parameters and be overloaded
- Provides better diagnostic at the point of use
- Cannot form partial statements or ”token soup”, which makes it possible to provide good tooling and limit the potential for exotic and clunky DSL.

Comparison with Parametric Expressions

The facilities presented here are very similar to the Parametric Expression proposal [3], with two exceptions:

- They are composed of features otherwise offered by reflection, reification and code injection
- Both expressions and complete statements can be injected.

Purposefully, neither proposal supports partial statement or partial expression, which would hinder tooling, maintainability and diagnostics.

Examples

[P1221] [3] offers many great example. For comparison, here is how the same could be achieve using the facilities we present:

if

```
consteval void constexpr_if(constexpr bool condition,
                   meta::any_expression auto a,
                   meta::any_expression auto b) {
    if constexpr(condition){
        -> exprid(a);
    }
    else {
        -> exprid(b);
    }
}

int main() {
    constexpr_if(
       true,
       reflexpr(print(”Hello, world!”)),
        reflexpr(print("not evaluated so nothing gets printed"))
    );
}
```

The above example makes use of `constexpr` parameters.

`fwd`

```
consteval void fwd(meta::any_expression auto x) {
    -> static_cast<decltype(x)>(exprid(x));
}
auto new_f = [](auto&& x) { return fwd(reflexpr(x)); };
```

`make_array`

```
consteval void make_array(meta::any_expression auto... x) {
    -> std::array<std::common_type<typeof(x)...>, sizeof...(x)>{exprid(x)...};
}
```

`push_back` (from P0927)

This example inspired by P0927 [1] illustrate some un-answered questions and limitations of this proposal:

- What about member access when the `consteval` function is defined within a class?
- Injection creates code at the call site, while lazy evaluation introduces a new object.

  Pick your poison.

```
template <typename T>
consteval reference vector<T>::better_push_back(meta::any_expression auto... args) {
   -> { [this, &args]{
        __grow_if_needed();
        new (&__m_array[__m_size]) T(exprid(args)...);
        _return __m_size[__m_size++];
     }();
   };
}
std::vector<std::vector<int>> v1;
v1.better_push_back(reflexpr{ 1, 2, 3 });
```

`std::assume`

P1774 [2] proposes `std::assume`, which could be implemented as

```
consteval void assume(expression<bool> auto expr) {
    -> __builtin_assume(exprid(expr));
}
```

This alleviates the need for making it a magic function or to introduce a new language syntax.

`std::assert`

```
consteval void assert(std::expression<bool> auto expr, std::string_view msg) {
#ifdef _DEBUG
    -> { [&expr, &msg]{
            if(!exprid(expr)) {
                std::format("assert: {} failed : {}", meta::to_string(expr), msg);
            }
        }();
    };
#endif
}
```

ABSL_FLAG

The Abseil library has a command-line flags feature whose primary API is the `ABSL_FLAG` macro. The following examples are taken from the abseil website:

```
ABSL_FLAG(bool, big_menu, true,
    "Include 'advanced' options in the menu listing");
ABSL_FLAG(std::string, output_dir, "foo/bar/baz/", "output file dir");
ABSL_FLAG(std::vector<std::string>, languages,
    std::vector<std::string>({"english", "french", "german"}),
    "comma-separated list of languages to offer in the 'lang' menu");
ABSL_FLAG(absl::Duration, timeout, absl::Seconds(30), "Default RPC deadline");
```

Instead of a macro, we can use a `consteval` function:

```
template <typename T>
consteval void RegisterFlag(std::string_view name,
                            expression<T> default_value,
                            std::string_view help) {
    -> namespace :: {
        void* unqualid("AbslFlagsInitFlag", name) {
            return absl::flags_internal::MakeFromDefaultValue<T>(exprid(default_value));
        }
        constinit absl::Flag<Type> unqualid("ABSEIL_FLAGS_", name) {
            name,
            source_location::current().file_name(),
            &unqualid("AbslFlagsInitFlag", name),
            help
        };
    };
}
```

This implementation is incomplete compared to Abseil's. Regardless:

- It is much shorter as it does not have to deal with the idiosyncrasies of a preprocessor
- It does not present a DSL: this is regular c++ for which it is easy to provide completion and other tooling.
- Types are adequately checked
- It can be called from any namespace and will still inject the required symbols in the global namespace.

For the specific of `unqualid` and namespace injection, please refer to Andrew Sutton's P1717 [5]. This example is mostly provided to show that this feature is not limited to lazy evaluation and how the preprocessor # and ## operators can be replaced.

Expression decomposition and Unit Test frameworks

Unit test frameworks usually provide features which are cumbersome to use, unless making heavy use of the pre-processor.

```
    REQUIRE(expression == expected_result);
REQUIRE_EQ(expression, expected_result);
```

These macros, which vary slightly in name or behavior depending on the framework used, will usually evaluate each expression and print the expression and its value when the test fails. An egregious simplification of the implementation of such macro would look like:

```
template <typename T>
bool test(std::string_view str, T&& x) {
    const auto res = x;
    if(!res) {
        std::cout << str << ": Failed\n";
    }
    return res;

#define REQUIRE(...) test(#__VA_ARGS__, __VA_ARGS__)
int main() {
    REQUIRE(1 + 1 == 2);
}
```

Instead, using reflection and injection we can write the following function which is scoped and check the type of its parameters, providing better diagnostic:

```
consteval void REQUIRE(expression<bool> auto expr) {
    -> [expr] {
        const auto value = exprid(expr);
        if(!value) {
            show_error(meta::to_string(expr));
        }
    }();
}
```

We could further refine this idea by providing tools to decompose expressions into their composing sub-expressions, which in the present case may help provide better messages in our test framework.

This idea is somewhat orthogonal to the matter at hand and can be explored separately.

```
consteval void REQUIRE(expression<bool> auto expr) {
    -> [expr] {
        const auto value = exprid(expr);
        if(!value) {
            if(meta::is_binary_expression(expr)) {
                show_error(meta::to_string(meta::lhs(expr)),
                           exprid(meta::lhs(expr)),
                           meta::binary_operator(expr),
                           meta::to_string(meta::rhs(expr)),
                           exprid(meta::rhs(expr))
                );
            }
        }
        else {
            show_error(meta::to_string(expr));
        }
    }();
}
```

Design points

Identifying methods doing code injection

The presented `consteval` functions can either:

- Return a value
- Inject an expression
- Inject one or more statements

Injected statements are usually valid in a specific scope (function, class, namespace) and may or may not be declarations. Should this different type of injection be visually distinguished? In general code, injection behaves very differently than a simple call and would benefit from being visible **at call site**.

Lazy evaluation, reflection, and sigils

When an expression may conditionally not be evaluated, it should be visible as that may otherwise lead to surprising results if the code is either conditionally evaluated or conditionally injected, especially for facilities like `log` or `assert`. We think there are a few options to be considered:

- Explicitely applies the `reflexpr` operator to every expression. For example `log(reflexpr("Hello"));` However, this may become overly verbose and cumbersome.

  We may consider a shorter keyword, for example `log($("Hello"));`

  While nicely explicit, trying to extend the basic source character has historically been doomed to fail. And it is possible that it would still be considered to verbose.

- Introduce a tool separate from `consteval` functions for the purpose of code injection, with a different syntax for declaration and use.

  ```
  consteval log!(expression<std::string_view> auto message) {
      -> do_log(exprid(message));
  };
  void f() {
      log!("Hello");
  }
  ```

  We use the strawman syntax `consteval` *identifier* `!(args)` to declare hygenic macros (It's not a bad word!) and *identifier!* to invoke/expand them.

  ```
  f(std::fwd!(args));
  std::assert!(1+1 == 2, "Maths are broken");
  catch2::require!(vec.empty());
  abseil::RegisterFlag<bool>!("ftime-trace", false, "Profile compile time");
  ```

  There is another bit of magic involved here: Expressions are automatically converted to a `std::meta::info`.

  This is much nicer to use, but requires new syntax and doesn't allow differentiating between parameters that will always be evaluated and those which won't.

  I think this general approach to the syntax at call site is the most sensible.

  [*Note:* Yes, this looks like Rust macros, as Rust's macros also use a sigil to differentiate themselves from normal rust functions, for the same reasons. However, rust macros operate on tokens, which is very different from what is discussed here. Exclamation mark `!` happens to be fairly readable, feel free to bikeshed. —*end note*]

Comparison of this proposal, Lazy evaluation, and Parametric Expressions

Lazy evaluation, Parametric Expressions and this proposal all gravitate around the same design space and try to solve the same problem from different angles.

We think this proposal is more generic and simpler as it adds less new language features and simply evolves and refines proposed and planned features.

We leverage reflection and code injection to provide a feature akin to AST-level hygienic macro, which offer a lot of consistency and synergy with these other features, the sum of which would be able to replace all function macro use cases. Notably, this proposal allows the expansion to both expressions and statements.

All of that comes, of course at a price. Code Injection is not yet a mature proposal and represents a rather large body of work, that may not be standardized any time soon. In this regard, both lazy evaluation and parametric expressions are simpler because they have no prerequisite and could be standardized faster.

Lazy Evaluation creates an object for every expression and might perform a call. On the other hand, a function taking a lazily evaluated parameter can be defined in a separate translation unit and may generate less code.

Parametric Expressions are rather similar to the current proposal with a more narrow focus.

`constexpr` parameters

Some capabilities and example presented, notably `constexpr_if` may require a feature similar to `constexpr` parameters [4]. While there is some overlap between the problem that either proposal can solve and they maybe mutually beneficial, there are mostly orthogonal concerns.

Acknowledgments

This paper draws *heavily* on the work made on code injection and reflection, notably by Daveed Vandevoorde, Andrew Sutton, Faisal Vali, Wyatt Childers and Herb Sutter.

References

[1] James Dennett and Geoff Romer. P0927R2: Towards a (lazy) forwarding mechanism for c++. https://wg21.link/p0927r2, 10 2018.

[2] Timur Doumler. P1774R0: Portable optimisation hints. https://wg21.link/p1774r0, 6 2019.

[3] Jason Rice. P1221R1: Parametric expressions. https://wg21.link/p1221r1, 10 2018.

[4] David Stone. P1045R1: constexpr function parameters. https://wg21.link/p1045r1, 9 2019.

[5] Andrew Sutton and Wyatt Childers. P1717R0: Compile-time metaprogramming in c++. https://wg21.link/p1717r0, 6 2019.

[6] Andrew Sutton and Herb Sutter. P0712R0: Implementing language support for compile-time programming. `https://wg21.link/p0712r0`, 6 2017.

[7] Daveed Vandevoorde, Wyatt Childers, Andrew Sutton, Faisal Vali, and Daveed Vandevoorde. P1240R1: Scalable reflection in c++. `https://wg21.link/p1240r1`, 10 2019.

[8] Daveed Vandevoorde and Louis Dionne. P0633R0: Exploring the design space of metaprogramming and reflection. `https://wg21.link/p0633r0`, 3 2017.