Function Parameter Reflection in Reflection for C++26

Document #: P3096R2
Date: 2024/07/16
Project: Programming Language C++
Audience: Evolution Working Group
Library Evolution Working Group
Reply-to: Adam Lach
<adamlach@gmail.com>
Walter Genovese
<wgenovese@bloomberg.net>

Contents

1 Abstract

The goal of this paper is to motivate the need for and propose adding parameter reflection to the reflection proposal for C++26 [@].

2 Revisions

2.1 R0

Initial Version

2.2 R1

- added synopsis of the proposed API
- added proposed wording
- added poll results
- clarified the intended solution
- improved wording
- added SG7 poll results from Tokyo

2.3 R2

- added EWG poll results from St Louis
- improved proposed wording
- updated examples to use `expand` instead of `template for`
- added implementation experience notes

3 Motivation

Based on a few demonstrative applications, we claim that parameter reflection is an important feature that should be included in the initial specification for Reflection in C++26.

4 Use cases for parameter reflection

In this section, we aim at presenting a few important use cases for parameter reflection. We split them into reflecting parameter types and parameter names to demonstrate that having just parameter type reflection, which is well defined and should not spark any controversies, is very useful while adding parameter name reflection presents some ambiguities that could be faced in various ways. In many of the code examples we utilize the `expand` helper described in [P2996R3].

4.1 Dependency Injection / Inversion of Control

One of the known implementations of the Inversion of Control design pattern is the so-called Dependency Injection (DI). The idea behind DI is to preconfigure a builder or a factory with a set of dependencies that are then injected into the objects created by that builder or factory. In other languages, this is frequently accomplished through constructors, but in C++ it is not currently possible without additional scaffolding. However, we can use constructor parameter reflection to mitigate that shortcoming.

For the sake of simplicity and clarity, we made the following assumptions:

- there is only one non-default, non-copy and non-move constructor for the type T we are going to build using the builder
- the only constructor of the type T has the parameters that are injected in the builder appearing before the parameters that are not injected in the builder
- to inject dependency in the builder (also called service) we use the method `add_service()` of the builder
- to construct an object of type T via builder we call the method `build()` of the builder passing the arguments that were not added to the builder via the `add_service()` method

4.1.1 Type Based Dependency Injections

From the user perspective, the code could look like this:

```
struct Logger {...};
Logger makeLogger() {...}

struct Database {...};
Database makeDatabase() {...}

struct MyStruct {
    MyStruct(Logger const& logger, Database const& db, string const& s, int i);
};

struct YourStruct {
    YourStruct(Logger const& logger, int i);
};


// somewhere else in the code
DependencyInjectorByTypeBuilder dib;
dib.add_service(makeLogger())
   .add_service(makeDatabase());

// some other code

auto ms = dib.build<MyStruct>("s", 3);
auto ys = dib.build<YourStruct>(3);
```

See https://godbolt.org/z/azoT9Pb46 for the implementation of the `DependencyInjectorByTypeBuilder` class and the client code.

The key elements are the usage of `members_of()`, `is_constructor()`, `is_special_member()`, and `parameters_of()` to detect a constructor that is not a copy or move constructor and the usage of `type_of()` to get the type of the parameters.

4.1.2 Name-based Dependency Injections

To demonstrate the usefulness of reflecting on parameters' names in the context of dependency injection, we imagine a situation where multiple services with the same type are added to the builder (in our example, we chose dynamic polymorphism). From the user perspective, the code could look like this:

```
struct Database {...};

struct FastDb : public Database {...};

struct BasicDb : public Database {...};

struct MyStruct {
    MyStruct(Database const& primary, Database const& secondary, string const& s, int i);
};

struct YourStruct {
    YourStruct(Database const& primary, int i);
};


// somewhere else in the code
DependencyInjectorByNameBuilder<Database> ib;
ib.add_service("primary", make_unique<FastDb>())
  .add_service("secondary", make_unique<BasicDb>());

// some other code

auto ms = ib.build<MyStruct>("s", 3);
auto ys = ib.build<YourStruct>(3);
```

See https://godbolt.org/z/4To8YWfEe for the implementation `DependencyInjectorByNameBuilder` class and the client code.

The key element that is different compared to the previous case (parameter type reflection) is the use of `meta::name_of()` to get the name of the parameter and the usage of `expand()` (that we had to implement since it is not available in the experimental clang fork).

4.2 Language Bindings

Python Bindings for value-based reflection has been discussed extensively in [P2911R1]. In the conducted research, parameter reflection most notably proved to be indispensable in order to:

- generate bindings for constructors, and
- add keyword arguments support.

We repeat some of the discussion and examples here for the sake of completeness. The examples utilize pybind11 and are creating bindings of the following class:

```
struct Execution {
    enum class Type { new_, fill, ... }
    Execution(Order order, Type type);
    Execution(Order order, Type type,
              double price, size_t quantity = 0);
};
```

4.2.1 Constructor Bindings

While reflecting on parameter types of free functions / member functions is possible in C++ — even without reflection — the same is not possible for constructors. At the very least, support for parameter type reflection is therefore critical for this use case so it is possible to bind constructors without the need to spell out all constructor parameter types by hand.

Binding automation for class constructors using parameter type reflection:

```
template<typename Scope>
void bind_class(Scope&& scope) {
    // bind ctors
    [:expand(members_of(^func)):] >> [&]<auto e>{
       if constexpr (is_public(e) && is_constructor(e) &&
                     !is_copy_constructor(e) &&
                     !is_move_constructor(e)) {
           constexpr auto params = parameters_of(e);
          scope.def(py::init<...typename [:type_of(params):]...>());
        }
    };
   // ...
}
```

With that the binding code can be substantially simplified

Before

```
py::class_<Execution> scope(m, "Execution");
scope.def(py::init<Order, Execution::Type>());
scope.def(py::init<Order, Execution::Type,
                   double, size_t>());
```

After

```
py::class_<Execution> scope(m, "Execution");
bind_class(scope);
```

4.2.2 Keyword Arguments

Furthermore, parameter name reflection would allow for adding keyword arguments to function bindings. While the lack of that feature does not render the resulting Python module useless, it is certainly a major annoyance to Python users who are used to having keyword arguments support available by default in all functions.

Binding automation for class constructors with keyword arguments support using parameter name reflection:

```
template<typename Scope>
void bind_class(Scope&& scope) {
    // bind ctors
    [:expand(members_of(^func)):] >> [&]<auto e>{
       if constexpr (is_public(e) && is_constructor(e) &&
                     !is_copy_constructor(e) &&
                     !is_move_constructor(e)) {
            constexpr auto params = parameters_of(e);
            scope.def(py::init<...typename [:type_of(params):]...>(),
                      py::arg(...name_of(^[:params:]))...);
       }
    }
    //...
}
```

With that the binding code can be substantially simplified

Before

```
py::class_<Execution> scope(m, "Execution");
scope.def(py::init<Order, Execution::Type>(),
          py::arg("order"),
          py::arg("type"));
scope.def(py::init<Order, Execution::Type, double, size_t>(),
          py::arg("order"),
          py::arg("type"),
          py::arg("price"),
          py::arg("quantity") = 0);
```

After

```
py::class_<Execution> scope(m, "Execution");
bind_class(scope);
```

4.3 Debugging / Logging

This is a very simple use case of reflection utilisation, but it is one that has the potential to simplify logging substantially, especially in the presence of complex parameter types.

The following demonstrates a very simple way to log all function parameters and their values:

```
void func(int counter, float factor) {
    [:expand(parameters_of(^func)):] >> [&]<auto e> {
        cout << meta::name_of(e) << ": " << [:e:] << ", ";
    };
}


int main() {
    func(11, 22);
    func(33, 44);
}
```

See https://godbolt.org/z/frE7fzP1G.

5 Reflecting on Parameter Types

As showcased in the examples above, reflecting on parameter types is a feature that has many important applications and does not appear to carry any additional risks. We therefore propose to include it in the "Reflection for C++26" proposal. It is however necessary to address some corner cases.

5.1 Array to pointer decay

```
void fun(int arr[10]);
void fun(int* ptr);
```

These two function declarations are effectively identical. We therefore propose that `type_of` for function parameter reflections returns the effective type which is `int*`.

5.2 Type aliases

```
using int_ptr = int*;

void fun(int_ptr ptr);
void fun(int* ptr);
```

These two function declarations are identical. We therefore propose that `type_of` for function parameter reflections returns the effective type which is `int*`.

5.3 const and volatile qualification

Qualifying a function parameter's type with const and/or volatile does not introduce a new overload. (Note that int const& and int& are different types and not the same type with and without const qualification.) Therefore the following declarations refer to the same `func` overload.

```
int fun(int a);
int fun(int const a);
int fun(int volatile a);
int fun(int const volatile a);
int fun(int a) { return 42; }
```

Since effectively all of the above are the same and const and/or volatile have no effect in non-definition context we propose that for the sake of simplicity and consistency the cv-ness of parameters is not recorded in the reflection of parameter type.

```
meta::info param_type = type_of(parameters_of(^func)[0]);
static_assert(is_const(param_type) == false);
static_assert(is_const_v<typename [:param_type:]> == false);
```

6 Reflecting on Parameter Names

While very useful, reflecting on parameter names introduces a lot of nuanced problems that need to be addressed.

6.1 Problem Statement

How can parameter name reflection be utilized safely to implement generic / reusable libraries given that it is permissible for the same function to have its declarations specify different parameter names?

Using the lock3 implementation of [P1240R2] to demonstrate the error prone behavior of `name_of` (see https://godbolt.org/z/M84Ea6P88).

```
#include <experimental/meta>
#include <iostream>

using namespace experimental::meta;

// function declaration 1
void func(int a, int b);

void print_after_fn_declaration1() {
    cout << "func param names are: ";
    template for (constexpr auto e : parameters_of(^func)) {
        cout << name_of(e) << ", ";
    }
    cout << "\n";
}

// function declaration 2
void func(int c, int d);

void print_after_fn_declaration2() {
    cout << "func param names are: ";
```

```
    template for (constexpr auto e : parameters_of(^func)) {
        cout << name_of(e) << ", ";
    }
    cout << "\n";
}

// function definition
void func(int e, int f)
{
    return;
}

void print_after_fn_definition() {
    cout << "func param names are: ";
    template for (constexpr auto e : parameters_of(^func)) {
        cout << name_of(e) << ", ";
    }
    cout << "\n";
}

struct X {
    void mem_fn(int g, float h);
};

void print_after_mem_fn_declaration() {
    cout << "mem_fn param names are: ";
    template for (constexpr auto e : parameters_of(^X::mem_fn)) {
        cout << name_of(e) << ", ";
    }
    cout << "\n";
}

void X::mem_fn(int i, float j) {
    (void)i;
    (void)j;
}

void print_after_mem_fn_definition() {
    cout << "mem_fn param names are: ";
    template for (constexpr auto e : parameters_of(^X::mem_fn)) {
        cout << name_of(e) << ", ";
    }
    cout << "\n";
}

void print_class_members_after_definition() {
    template for (constexpr auto e : members_of(^X)) {
        cout << name_of(e) << " param names are: ";
        template for (constexpr auto p : parameters_of(e)) {
            cout << name_of(p) << ", ";
        }
        cout << "\n";
    }
```

```
}

int main() {
    print_after_fn_declaration1();          // prints: func param names are: a, b,
    print_after_fn_declaration2();          // prints: func param names are: c, d,
    print_after_fn_definition();            // prints: func param names are: e, f,
    print_after_mem_fn_declaration();       // prints: mem_fn param names are: g, h,
    print_after_mem_fn_definition();        // prints: mem_fn param names are: i, j,
    print_class_members_after_definition(); // prints: mem_fn param names are: g, h,
}
```

We can observe the following set of properties of the reflected names which make it hard for parameter reflection to be used safely to implement reusable library functions:

- they depend on the order of declarations
- they cannot be checked for consistency
- they depend on the way in which reflection is done; compare `print_after_mem_fn_definition` (direct reflection of `X::mem_fn`) vs `print_class_members_after_definition` (indirect reflection of `X::mem_fn`)

6.2 Ideal Solution

Based on the aforementioned properties and implementation feasibility, we propose that an ideal solution should have the following characteristics

- **consistent**: behaves consistently in all contexts (direct / indirect reflection)
- **order independent**: is not affected by changing the order of reachable declarations (and what implies changing the order of includes)
- **immediately applicable**: can work with existing code bases with minimal to no changes
- **self-contained**: requires minimal to no changes to the language outside of reflection
- **robust**: provides means of detecting name inconsistencies

We recognize that most of the solutions can benefit from the usage of external tooling like clang-tidy (readability-named-parameter), and that it should be recommended as best practice. However mandating specific tooling to be used for correct functioning of a library is far from ideal.

6.3 Considered Solutions

Based on our own research and gathered feedback, we present the following potential solutions to the problem.

6.3.1 No Guarantees

The simplest approach is to provide no guarantees at all. This means that whatever the compiler implementers decide is the most relevant function declaration in any given context, the parameter names of that function will be provided. This seems to be the approach taken in [P1240R2], likely leading to the inconsistencies we presented in the "Problem Statement". Using clang-tidy will not help resolve all of the inconsistencies, since it allows omitting parameter names in the definitions.

Solution characteristics:

- **consistent**: no
- **order independent**: no
- **immediately applicable**: yes
- **self-contained**: yes
- **robust**: no

6.3.2 Improved Consistency

It is possible to significantly improve on the “No Guarantees” approach by ensuring consistent behavior for direct and indirect reflections and by introducing a helper metafunction `has_consistent_paramater_names()`. The latter could help library implementers detect inconsistencies and warn the user and/or disable a feature which uses parameter names reflection.

Solution characteristics:

- **consistent**: yes
- **order independent**: no
- **immediately applicable**: yes
- **self-contained**: yes
- **robust**: yes

6.3.3 Enforced Consistent Naming

If there are multiple reachable declarations that have different parameter names — excluding omitted parameter names — parameter name reflection is invalid. Otherwise, the name returned for each parameter is its name found across all reachable declarations.

Assuming that

- `parameters_of()` accepts a reflection of a function and returns a range of info objects representing the reflections of its function parameters and
- `names_of()` is implemented as

```
template<info I>
std::vector<std::string_view> names_of() {
    auto res = std::vector<std::string_view>{};
    [:expand(parameters_of(I)):] >> [&]<auto e> {
        res.push_back(name_of(e));
    };
    return res;
}
```

enforced consistent naming should behave like the following

```
void func(int a, float b);
void func(int a, float c);

names_of(parameters_of(^func)); // fails to be a constant expression because the second parameter has inc
```

See https://godbolt.org/z/zrx3oMToM

```
void func(int, float b);
void func(int a, float);

names_of(parameters_of(^func)); // yields ["a", "b"]
```

See https://godbolt.org/z/1v8GcEPzh

```
void func(int, float);
void func(int a, float);

names_of(parameters_of(^func)); // yields ["a", ""]
```

See https://godbolt.org/z/q6sfda4aE

```
void func(int a, float b);
void func(int, float);

names_of(parameters_of(^func)); // yields ["a", "b"]
```

See https://godbolt.org/z/hr9c5EcMf

Many code bases which utilize almost consistent parameter naming - no name or otherwise consistent names - will be able to utilize parameter name reflection without any changes. This is especially true, as this approach is consistent with the readability-named-parameter of clang-tidy.

Solution characteristics:

- **consistent**: yes
- **order independent**: yes
- **immediately applicable**: partially
- **self-contained**: yes
- **robust**: yes

6.3.4 Language Attribute

Another approach would be to introduce a new language attribute like `[[canonical]]`. If any reachable function declaration has this attribute, parameter name reflection always returns the names of parameters of that declaration. Otherwise, reflecting on parameter names is invalid. Only one function declaration with the `[[canonical]]` attribute is allowed to be reachable from any context.

```
template<meta::info I>
void print_param_names() {
    [:expand(parameters_of(I)):] >> [&]<auto e> {
        cout << meta::name_of(e) << ", ";
    };
}

[[canonical]]
void func(int a, float b);

void func(int x, float y) {
    [:expand(parameters_of(^func)):] >> [&]<auto e> {
        cout << meta::name_of(e) << ": " << [:e:] << ", ";
    };
}

int main() {
    print_param_names<func^>(); // prints "a, b,"
    func(33, 44);               // prints "a: 33, b: 44," instead of "x: 33, y: 44,"
}
```

The weakness of this approach is in those use cases where a function reflects on itself (see logging use case above). The list of parameter names might be different than the ones spelled out in its signature (the canonical declaration might have different ones) leading to unintuitive results.

Solution characteristics:

- **consistent**: yes
- **order independent**: yes
- **immediately applicable**: no
- **self-contained**: no
- **robust**: yes

6.3.5 User Defined Attribute

Yet another approach would be to utilize a user-defined attribute to mark the canonical function, such as in the code below:

```
// A declaration of function foo.
void foo(int n, int m);

// Another declaration of function foo, with the special attribute.
[[refl_bind::infer_param_names]]
void foo(int apples, int bananas);

// The definition of foo.
void foo(int a, int b) {
    return a + b;
}
```

Using user-defined attributes like this would require a more complex way of performing reflection of parameter names than what [P1240R2] proposed. In order for any code to identify the parameter names of a declaration annotated with `[[refl_bind::infer_param_names]]`, the following would be needed:

- the support for user defined attributes in the language, and
- a reflection facility to get (user defined) attributes, and
- a reflection facility to get *all* reachable declarations (with their parameters and attributes)

Solution characteristics:

- **consistent**: yes
- **order independent**: yes
- **immediately applicable**: no
- **self-contained**: no
- **robust**: yes

6.4 Solutions Summary

	consistent	**order independent**	**immediately applicable**	**self-contained**	**robust**
No Guarantees	no	no	yes	yes	no
Improved Consistency	yes	no	yes	yes	yes
Enforced Consistent Naming	yes	yes	partially	yes	yes
Language Attribute	yes	yes	no	no	yes
User Defined Attribute	yes	yes	no	no	yes

It seems to us that the “Enforced Consistent Naming” has the best characteristics and has the added benefit of being consistent with the clang-tidy readability-named-parameter check.

7 Reflecting on Parameter Default Values

For completeness sake, we are mentioning default values for parameters. In our research, we have encountered the need to know whether a parameter has a default value, but not necessarily what the value is. Since the language forbids redeclaration of the same function with a default value for the same parameter, there is no ambiguity. Therefore, we only propose to add the `has_default_argument` as it was already in [P1240R2].

8 ODR Violations

Reflection on function parameters depends on which function declarations are reachable in the reflection context. Due to that there is a possibility of introducing ODR violations. However, this problem does not pertain only to function parameters reflection, but also to reflecting namespaces or incomplete types. Therefore, we do not attempt to propose any solutions in this document, but defer to the authors of [P2996R2] for a more generic solution of the problem.

9 Proposed Metafunctions

9.1 Synopsis

```
namespace meta {
  consteval auto parameters_of(info r) -> vector<info>;
  consteval auto has_consistent_name(info r) -> bool;
  consteval auto name_of(info r) -> std::string_view;
  consteval auto u8name_of(info r) -> std::u8string_view;
  consteval auto any_name_of(info r) -> std::string_view;
  consteval auto u8any_name_of(info r) -> std::u8string_view;
  consteval auto has_ellipsis_parameter(info r) -> bool;
  consteval auto has_default_argument(info r) -> bool;
  consteval auto is_explicit_object_parameter(info r) -> bool;
  consteval auto is_function_parameter(info r) -> bool;
  consteval auto return_type_of(info r) -> info;
  consteval auto type_of(info r) -> info;
}
```

9.2 parameters_of

```
namespace meta {
  consteval auto parameters_of(info r) -> vector<info>;
}
```

Given a reflection `r` that designates a function or a member function, return a vector of the reflections of the explicit parameters.

9.3 has_consistent_name

```
namespace meta {
  consteval auto has_consistent_name(info r) -> bool;
}
```

Given a reflection `r` that designates a function parameter, return `true` if the names of that parameter are the same or empty in all reachable declarations. Otherwise `false`.

9.4 name_of

```
namespace meta {
  consteval auto name_of(info r) -> std::string_view;
  consteval auto u8name_of(info r) -> std::u8string_view;
}
```

Given a reflection `r` that designates a function parameter, for which `has_consistent_name` evaluates to `true`, return a `string_view` or a `u8string_view` holding the non-empty name of that parameter. If `r` designates a function parameter, for which `has_consistent_name` evaluates to `false` then `name_of` and `u8name_of` are not constant expressions. Otherwise return the unqualified name of the reflected entity.

9.5 any_name_of

```
  consteval auto any_name_of(info r) -> std::string_view;
  consteval auto u8any_name_of(info r) -> std::u8string_view;
```

Given a reflection `r` that designates a function parameter, return the name of that parameter from any of the reachable declarations. Otherwise return the result of `name_of`.

9.6 has_ellipsis_parameter

```
namespace meta {
  consteval auto has_ellipsis_parameter(info r) -> bool;
}
```

Given a reflection `r` that represents a function, return `true` if that function or function type terminates with an ellipsis. Otherwise `false`.

9.7 has_default_argument

```
namespace meta {
  consteval auto has_default_argument(info r) -> bool;
}
```

Given a reflection `r` that designates a function parameter, return `true` if that parameter has a default value in any of the reachable declarations. Otherwise `false`.

9.8 is_explicit_object_parameter

```
namespace meta {
  consteval auto is_explicit_this_parameter(info r) -> bool;
}
```

Given a reflection `r` that designates a function parameter, return `true` if the parameter is referring to an explicit `this` parameter. Otherwise `false`.

9.9 is_function_parameter

```
namespace meta {
  consteval auto is_function_parameter(info r) -> bool;
}
```

Given a reflection `r`, return `true` if that reflection designates a function parameter. Otherwise `false`.

9.10 return_type_of

```
namespace meta {
  consteval auto return_type_of(info r) -> info;
}
```

Given a reflection `r` that designates a function or a function type, return a reflection that designates the return type of that function after applying array to pointer decay, expanding aliases and removing `const` and `volatile`.

9.11 type_of

```
namespace meta {
  consteval auto type_of(info r) -> info;
}
```

Given a reflection `r` that designates a function parameter, return the reflection of the type of the parameter after applying array to pointer decay, expanding aliases and removing `const` and `volatile` qualification.

10 Proposed Wording

10.1 library

10.1.1 [meta.reflection.names] Reflection names and locations

```
consteval auto any_name_of(info r) -> std::string_view;
consteval auto u8any_name_of(info r) -> std::u8string_view;
```

Constant When: If returning `string_view`, the unqualified name is representable using the ordinary string literal encoding. `r` represents a declared entity that is not a parameter of a function for which any two reachable declarations have a different non-empty name.

Returns: The unqualified name of the entity represented by `r`. If this entity has no name, then an empty `string_view` or `u8string_view`, respectively.

```
consteval auto name_of(info r) -> std::string_view;
consteval auto u8name_of(info r) -> std::u8string_view;
```

Constant When: If returning `string_view`, the unqualified name is representable using the ordinary string literal encoding. `r` represents a declared entity that is not a parameter of a function for which any two reachable declarations have a different non-empty name.

Returns: The unqualified name of the entity represented by `r`. If this entity is a function parameter then the first reachable non-empty name of that parameter. If this entity has no name, then an empty `string_view` or `u8string_view`, respectively.

10.1.2 [meta.reflection.queries] Reflection queries

```
consteval vector<info> parameters_of(info r);
```

Constant When: `is_function(r)` evaluates to `true`

Returns: A `vector` containing the reflections of all the explicit parameters of the function designated by `r` in the order in which they are declared.

```
consteval bool has_consistent_name(info r);
```

Returns: `true` if `r` represents a function parameter that has the same name or no name in all reachable declarations. Otherwise `false`.

```
consteval bool has_ellipsis_parameter(info r);
```

Returns: `true` if `r` represents a function that terminates with an ellipsis. Otherwise `false`.

```
consteval bool has_default_argument(info r);
```

Returns: `true` if `r` represents a function parameter that has a default value. Otherwise `false`.

```
consteval bool is_explicit_object_parameter(info r);
```

Returns: `true` if `r` represents a function parameter that is an explicit `this` parameter. Otherwise `false`.

```
consteval bool is_function_parameter(info r);
```

Returns: `true` if `r` represents a function parameter. Otherwise `false`.

```
consteval info return_type_of(info r);
```

Constant When: `r` is a reflection representing a function that is not a constructor or a destructor.

Returns: The reflection of the return type of the function represented by `r`. If that type is

- a type alias, then the reflection of the type underlying that alias
- an "array of T", then the reflection of a "pointer to T" type
- const qualified type, then the reflection of that type without const qualification
- volatile qualified type, then the reflection of that type without volatile qualification

```
consteval info type_of(info r);
```

Constant When: `r` represents a typed entity. `r` does not represent a constructor, destructor, or structured binding.

Returns: A reflection of the type of that entity.

If that entity is a function parameter declared with:

- a type alias, then the reflection of the type underlying that alias
- an "array of T", then the reflection of a "pointer to T" type
- const qualified type, then the reflection of that type without const qualification
- volatile qualified type, then the reflection of that type without volatile qualification

Otherwise, if every declaration of that entity was declared with the same type alias (but not a template parameter substituted by a type alias), the reflection returned is for that alias. Otherwise, if some declaration of that entity was declared with an alias it is unspecified whether the reflection returned is for that alias or for the type underlying that alias. Otherwise, the reflection returned shall not be a type alias reflection.

11 Implementation Experience

The enforced consistent naming as proposed has been implemented by Dan Katz in Bloomberg's open sourced clang fork.

- `name_of` https://godbolt.org/z/n9e7rb6oT, https://godbolt.org/z/E1M69GnWq
- `any_name_of` https://godbolt.org/z/bc73qoq1K
- `is_const` and `is_volatile` https://godbolt.org/z/1r1Kf3fd4, https://godbolt.org/z/b5rc7EKne
- `type_of`

    - arrays decay to pointer https://godbolt.org/z/YMTbTeGax
    - volatile and const are ignored https://godbolt.org/z/4ndbPsoMs
    - aliases are expanded https://godbolt.org/z/dK3qqMYxc
- `is_explicit_object_parameter` https://godbolt.org/z/xG5r7oahj
- `has_default_argument` https://godbolt.org/z/Yf4rahqdv
- `has_ellipsis_parameter` https://godbolt.org/z/xr4hfYnnd

12 Polls

12.1 P3096R0: SG7, March 2024, WG21 meeting in Tokyo

POLL: P3096R0 - Function Parameter Reflection in Reflection for C++26: We want this problem to be solved and we would like to see an updated paper (with wording) in EWG and LEWG.

SF: 5 F: 5 N: 0 A: 0 SA: 0

12.2 P3096R1: EWG, June 2024, WG21 meeting in St. Louis

Poll: P3096R1 - Function Parameter Reflection in Reflection for C++26, we are interested in this paper, encourage further work based on feedback provided, and would like to see it updated with Core wording expert feedback.

SF: 5 F: 13 N: 7 A: 0 SA: 0

13 Acknowledgments

We'd like to thank Daveed Vandevoorde, Sergei Murzin, Andrea Chiarini, Jagrut Dave, Dan Katz, Inbal Levi, Marios Katsigiannis, Patrick Martin, Vittorio Romeo, David Sankel, Mark Sciabica, Saksham Sharma, and Andrei Zissu for their valuable insights and support in shaping this document.

14 References

[P1240R2] Daveed Vandevoorde, Wyatt Childers, Andrew Sutton, Faisal Vali. 2022-01-14. Scalable Reflection.
https://wg21.link/p1240r2

[P2911R1] Adam Lach, Jagrut Dave. 2023-10-13. Python Bindings with Value-Based Reflection.
https://wg21.link/p2911r1

[P2996R2] Barry Revzin, Wyatt Childers, Peter Dimov, Andrew Sutton, Faisal Vali, Daveed Vandevoorde, Dan Katz. 2024-02-15. Reflection for C++26.
https://wg21.link/p2996r2

[P2996R3] Barry Revzin, Wyatt Childers, Peter Dimov, Andrew Sutton, Faisal Vali, Daveed Vandevoorde, Dan Katz. 2024-05-22. Reflection for C++26.
https://wg21.link/p2996r3