Function Parameter Reflection in Reflection for C++26

Document #:	P3096R0
Date:	2024/02/14
Project:	Programming Language C++
Audience:	SG7 Reflection Study Group

Reply-to:	Adam Lach <alach3@bloomberg.net> 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 [P2996R1].

2 Revisions

2.1 R0

Initial Version

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. Any code examples using function parameter reflection are, by necessity, based on [P1240R2].

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
- calling T(S0 s0, ..., SM sm, A0 a0, ..., AN an) is unambiguous, where
  - T denotes the type of the constructed object and,
  - s0, ..., sM are service instances injected to the builder and
  - a0, ..., aN are parameters forwarded to T via `build(...)`

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, std::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/K537Wor9s for the implementation of the `DependencyInjectorByTypeBuilder` class and the client code.

Note that, in some cases, we had to implement workarounds to problems of the experimental implementation. For example, meta::substitute seemed not to work.

Besides the workarounds, the key elements are the following:

— the usage of `template for()` and `meta::parameters_of()` to go through all the parameters of a constructor, and for each parameter the usage of `meta::type_of()` to get the type of it
— the usage of `template for()`, `meta::members_of()`, `meta::is_constructor()`, `meta::is_copy_constructor()`, and `meta::is_move_constructor()` to detect a constructor that is not a copy or move constructor.

4.1.2 Name-based Dependency Injections

To demonstrate how useful it could be to access parameters' names in this use case, we imagine a situation where we can add services to the builder that always have the same type (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, std::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", std::make_unique<FastDb>())
  .add_service("secondary", std::make_unique<BasicDb>());

// some other code

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

See https://godbolt.org/z/o14ddvqb8 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.

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:

```
auto scope = py::class_<Execution>(module, "Execution");

template for (constexpr auto e : member_fn_range(^Execution)) {
    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):]...>());
    }
}
```

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:

```
auto scope = py::class_<Execution>(module, "Execution");

template for (constexpr auto e : member_fn_range(^ClassT)) {
      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:]))...);
      }
}
```

4.3 Debugging / Logging

This is a very simple use case of reflection utilization, 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) {
    template for (constexpr auto e : meta::parameters_of(^func))
        std::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.

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 and its definition specify different parameter names?

Using the lock3 implementation of [P1240R2] (see https://godbolt.org/z/M84Ea6P88)

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

using namespace std::experimental::meta;

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

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

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

void print_after_fn_declaration2() {
    std::cout << "func param names are: ";
    template for (constexpr auto e : param_range(^func)) {
        std::cout << name_of(e) << ", ";
    }
    std::cout << "\n";
```

```
}

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

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

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

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

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

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

void print_class_members_after_definition() {
    template for (constexpr auto e : member_fn_range(^X)) {
        std::cout << name_of(e) << " param names are: ";
        template for (constexpr auto p : param_range(e)) {
            std::cout << name_of(p) << ", ";
        }
        std::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 or definition 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. It has to be noted, however, that unless carefully specified, using clang-tidy will not help resolve all of the inconsistencies, since it allows omitting parameter names in the definitions.

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 or definitions 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 or definitions.

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

names_of(parameters_of(^func)); // yields an error
```

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

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

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

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

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

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

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 or definition has this attribute, parameter name reflection always returns the names of parameters of that declaration or definition. Otherwise, reflecting on parameter names is invalid. Only one function declaration or definition with the `[[canonical]]` attribute is allowed to be reachable from any context.

```
template<meta::info I>
void print_param_names() {
    template for (constexpr auto e : meta::parameters_of(I))
```

```
        std::cout << meta::name_of(e) << ", ";
}

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

void func(int x, float y) {
    template for (constexpr auto e : meta::parameters_of(^func))
        std::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 or definition 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 and definitions (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 include the `has_default_argument` metafunction in the [P2996R1], as it was already in [P1240R2].

8 Our Proposal

We propose to amend [P2996R1] to include the following:

— reflection of function parameter types
— reflection of function parameter names based on the proposed "Enforced Consistent Naming" solution
— restore `has_default_argument` metafunction
— restore `is_function_parameter` metafunction

9 Acknowledgments

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

10 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

[P2996R1] Barry Revzin, Wyatt Childers, Peter Dimov, Andrew Sutton, Faisal Vali, Daveed Vandevoorde. 2023-12-18. Reflection for C++26. https://wg21.link/p2996r1