Function Parameter Reflection in Reflection for C++26

Document #: P3096R12
Date: 2025-06-20
Project: Programming Language C++
Audience: Core Language Wording
Library Wording
Reply-to: Adam Lach
<adamlach@gmail.com>
Dan Katz
<dkatz85@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 [P2996R12].

2 Revisions

2.1 R12

— changed the wording for identifier_of() and has_identifier() as per EWG poll
— removed constant when for has_default_argument() as per LEWG poll
— added poll results from LEWG on 2025-06-19
— added poll results from EWG on 2025-06-19

2.2 R11

— updated wording based on CWG feedback on 2025-06-18

2.3 R10

— aligned wording with [P2996R12]
— updated wording based on feedback from Dan Katz
— updated wording based on LWG feedback on 2025-06-17

2.4 R9

— updated wording based on CWG feedback

2.5 R8

— aligned wording with [P2996R11]

2.6 R7

— added EWG poll results from Hagenberg
— updated wording based on LWG feedback on 2025-02-11

2.7 R6

— added poll results from LEWG telecon of 2025-01-21
— updated wording based on LEWG feedback
— added sample code for dependency injection
— removed trailing return type in declarations

2.8 R5

— changed parameter_variable to variable_of
— aligned wording with [P2996R7]
— added poll results from Wrocław

2.9 R4

— improved problem statement
— added parameter_variable
— clarified the rationale for type_of

2.10 R3

- improved proposed wording
- aligned with [P2996R5]

2.11 R2

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

2.12 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

2.13 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. In many of the code examples we utilize the `expand` helper described in [P2996R12].

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/WhfWvW14n for the implementation of the `DependencyInjectorByTypeBuilder` class and the client code.

The `DependencyInjectorByTypeBuilder` has the following two public methods:

```
class DependencyInjectorByTypeBuilder {
vector<any> d_services;
public:
    template <typename S>
    DependencyInjectorByTypeBuilder& add_service(S&& s)
    {
        d_services.emplace_back(forward<S>(s));
        return *this;
    }
    template <typename T, typename ... Args>
    T build(Args&& ... args) const;


    // more code
};
```

The `build` method is the following (no checks of error conditions).

```
template <typename T, typename ... Args>
T DependencyInjectorByTypeBuilder::build(Args&& ... args) const {
    constexpr std::meta::info DICtor = dependencyInjectorByTypeConstructor(^^T);
    constexpr unsigned NumCtorParams = [] consteval {
        return parameters_of(DICtor).size(); }();
```

```
    constexpr unsigned NumMissingParams = NumCtorParams - sizeof...(args);
    using mp_type_list = typename [: missingParamsTypes(DICtor, NumMissingParams) :];
        return make<T>(getMissingArgs(mp_type_list{}),
                       forward<Args>(args)...);
}
```

The utility `dependencyInjectorByTypeConstructor` is used to get the reflection of the constructor from the type `T`.

```
consteval meta::info dependencyInjectorByTypeConstructor(meta::info T) {
    auto nonspecial_ctor_with_params = [](auto r) {
        return is_constructor(r) &&
               !is_special_member_function(r) && !parameters_of(r).empty();
    };
    auto const members = meta::members_of(T);
    auto ctorsWithParams = members | views::filter(nonspecial_ctor_with_params);
    return (1 == ranges::distance(ctorsWithParams))
         ? (*ranges::begin(ctorsWithParams))
         : meta::info{};
}
```

The utility `missingParamsTypes` is used to get a reflection of a type list that is then passed to `getMissingArgs` whose output is finally passed to `make`

```
consteval meta::info missingParamsTypes(meta::info M, unsigned N)
{
    return substitute(^^type_list,
                      parameters_of(M) |
                      views::transform(meta::type_of) |
                      views::take(N));
}


class DependencyInjectorByTypeBuilder {
  // more code
private:
  template <typename T, typename ... Ts, typename ... Args>
  T make(tuple<Ts...>&& tuple, Args&& ... args) const {
      return T{get<Ts>(tuple)..., forward<Args>(args)... };
  }

  template <typename T>
  T getArgByType() const {
    for (auto const& e : d_services) {
       if (auto const * ptr = any_cast<decay_t<T>>(&e)){
                return *ptr;
       }
    }
    throw logic_error("No service available!");
  }


  template <typename ... Ts>
  tuple<Ts...> getMissingArgs(type_list<Ts...> const&) const {
     return {getArgByType<Ts>()...};
```

```
  }
};
```

4.1.2 Name-based Dependency Injections

To demonstrate the usefulness of reflecting on parameters' names in the context of dependency injection, we consider a situation where multiple services with the same type are added to the builder. 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/7TcWvqWzn for the implementation `DependencyInjectorByNameBuilder` class and the client code.

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 functions is possible in C++ — even without reflection — the same is not possible for constructors. 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	After

`py::class_<Execution> scope(m, "Execution");`
`scope.def(py::init<Order, Execution::Type>());`
`scope.def(py::init<Order, Execution::Type,`
`                   double, size_t>());`
	`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(...identifier_of(^^[:params:]))...);
       }
    }
    //...
}
```

With that the binding code can be substantially simplified:

Before	After

`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);`
	`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 << identifier_of(e) << ": " << [:e:] << ", ";
    };
}


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

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

5 Problem Statement

How can parameter reflection be utilized safely to implement generic / reusable libraries given that it is permissible for the same function to be declared multiple times with different parameter names, differently qualified types and default arguments?

5.1 Inconsistent Parameter Name

We are using the lock3 implementation of [P1240R2] to demonstrate the error prone behavior of `name_of` (see https://cppx.godbolt.org/z/bqjeexqq5). Note that: - `name_of` has been renamed to `identifier_of` in [P2996R12] - the old syntax for reflection was using a single `^` - `template for` was available to iterate over ranges of reflections

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

using namespace std;
using namespace std::experimental::meta;

// function declaration 1
```

```cpp
void func(int const a, int b  = 10);

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

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

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

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

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

// member function declaration
struct X {
    void mem_fn(int g, float h);
};

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

// member function definition
void X::mem_fn(int i, float const j) {
    (void)i;
    (void)j;
}
```

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

void print_class_members_after_definition() {
    template for (constexpr auto e : member_fn_range(^X)) {
        cout << name_of(e) << " params are: ";
        template for (constexpr auto p : param_range(e)) {
            cout << name_of(type_of(p)) << " " << name_of(p) << ", ";
        }
        cout << "\n";
    }
}

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

We can observe a set of properties of the reflected names and types 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`)

As a result very simple code changes, routinely performed by programmers, may become a source of difficult to detect bugs. Some examples of that:

- Reordering of member function definitions
- Moving function definitions to .cpp
- Changing "direct" to "indirect" reflection
- Reordering of includes

5.2 Default arguments are merged

Default arguments might be specified across multiple declarations, which makes it harder to clearly define which declaration is canonical.

```
void fun(int x, int y, int z = 0);
void fun(int x, int y = 0, int const z);

void main() {
   fun(1);
}
```

5.3 Inconsistent Parameter CV-Qualification

The language is not consistent in the treatment of parameter cv-qualification.

```
void fun(int const t) {
    static_assert( same_as<decltype(t), int const> );
    static_assert( same_as<decltype(fun), void (int) > ); // type adjustment
}
```

6 Ideal Solution

Based on the aforementioned properties and implementation feasibility, we propose the following characteristics to guide validate the best approach for reflecting function parameters:

- **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 (e.g. 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.

7 Considered Solutions

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

7.1 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 and types (with their cv qualification) 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".

Solution characteristics:

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

7.2 Improved Consistency

It is possible to significantly improve on the "No Guarantees" approach by ensuring consistent behavior for direct and indirect reflections and for different cv qualification of the parameter types by introducing two helper metafunctions `has_identifier()` and `has_consistent_type()`. These metafunctions could help library implementers detect inconsistencies and warn the user and/or disable a feature which uses parameter names and types reflection.

The downside of this approach is that likely any usage of parameter name reflection, outside of trivial applications like diagnostics, would need to be guarded with a consistency check. Otherwise bugs could easily creep into the code base unnoticed.

Solution characteristics:

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

7.3 Enforced Consistency

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

The types of reflected function parameters are subject to type adjustment resulting in no inconsistencies.

Assuming that `parameters_of` accepts a reflection of a function and returns a range of info objects representing the reflections of its function parameters, the enforced consistency should behave as demonstrated below.

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

type_of(parameters_of(^^func)[1]); // yields ^^float
```

https://godbolt.org/z/68Tz37vWE

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

identifier_of(parameters_of(^^func)[0]); // yields "a"
identifier_of(parameters_of(^^func)[1]); // fails to be a constant expression because the second paramete
```

See https://godbolt.org/z/5Wzq4MEdr

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

identifier_of(parameters_of(^^func)[0]); // yields "a"
identifier_of(parameters_of(^^func)[1]); // yields "b"
```

See https://godbolt.org/z/1rxavE5Mc

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

identifier_of(parameters_of(^^func)[0]); // yields "a"
identifier_of(parameters_of(^^func)[1]); // fails to be a constant expression
```

See https://godbolt.org/z/GdGjKEh61

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

identifier_of(parameters_of(^^func)[0]); // yields "a"
identifier_of(parameters_of(^^func)[1]); // yields "b"
```

See https://godbolt.org/z/xxqvxaT9Y

Many code bases which utilize almost consistent parameter naming - unnamed 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

7.4 Language Attribute

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

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

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

void func(int x, float y) {
    [:expand(parameters_of(^^func)):] >> [&]<auto e> {
        cout << identifier_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 and types might be different than the ones spelled out in its signature (the canonical declaration might have different ones) leading to unintuitive results. In addition to that, this approach would not be intuitive in situations where default arguments are specified across multiple declarations.

Solution characteristics:

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

7.5 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_parameters]]
```

```
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 and types than what [P1240R2] proposed. In order for any code to identify the parameter names and types of a declaration annotated with `[[refl_bind::infer_parameters]]`, 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

7.6 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

Based on the above, the “Enforced Consistency” approach has the best characteristics.

8 Reflecting on Parameter Types

8.1 Array to Pointer Decay

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

These two function declarations are effectively identical.

8.2 Type Aliases

```
using int_ptr = int*;
```

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

These two function declarations are identical.

8.3 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 `fun` 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; }
```

8.4 Type Adjustment of parameter-type-list

The language already performs type adjustment on the parameter-type-list of function type

```
void fun(int const t) {
    static_assert( same_as<decltype(t), int const > );
    static_assert( same_as<decltype(fun), void (int) > ); // type adjustment
}
```

8.5 Splicing in Function Definition

We recognize that within function definitions it is desirable that

```
void fun(int const t) {
    static_assert(is_const_v([: type_of(parameters_of(^^fun)[0]) :]));
    static_assert(same_as<[: type_of(parameters_of(^^fun)[0]) :], [: type_of(^^t) :]>);
}
```

8.6 Proposal

We propose to extend the `type_of` metafunction specified in [P2996R12] to return the type adjusted parameter type (de-aliased, cv-unqualified, pointer-decayed). This is to achieve increased consistency of the results of `type_of` and consistency with function type.

```
int fun(int a, int b);
int fun(int const c, int b);

void reflect_fun() {
  static_assert(same_as<[: type_of(parameters_of(^^fun)[0])) :], int>);
  int vv = [: parameters_of(^^fun)[0] :]; // error
}
```

We also propose the addition of a `variable_of` metafunction returning the reflection of the exact parameter of the definition. This is to allow for splicing of parameter types in function definitions.

```
int fun(int a, int b);
int fun(int const c, int b) {
  static_assert(!is_const(type_of(parameters_of(^^fun)[0])));
  static_assert(parameters_of(^^fun)[0] != ^^c);
```

```
  static_assert(variable_of(parameters_of(^^fun)[0]) == ^^c);
  static_assert(is_const(
                variable_of(parameters_of(^^fun)[0])));
  static_assert(is_const_v<
                [: variable_of(parameters_of(^^fun)[0]) :]);
}
```

9 Reflecting on Parameter Names

9.1 Proposal

We propose to extend the `identifier_of` metafunction specified in [P2996R12] to return the name of a function parameter if all reachable function declarations have the same name for that parameter.

```
int fun(int a, int b);
int fun(int const a, int b);

void reflect_fun1() {
  assert(identifier_of(parameters_of(^^fun)[0]) == "a"sv);
}

int fun(int const c, int b);

void reflect_fun2() {
  auto vv = identifier_of(parameters_of(^^fun)[0]); // error
}
```

We also propose the addition of a `variable_of` metafunction returning the reflection of the exact parameter of the definition. This is to facilitate splicing of parameter names in function definitions in the presence of name inconsistencies.

```
int fun(int a, int b);
int fun(int const c, int b) {
  static_assert(variable_of(parameters_of(^^fun)[0]) == ^^c);
  assert(identifier_of(variable_of(parameters_of(^^fun)[0])) == "c"sv);
}
```

10 Splicing in Function Definition

We recognize that within function definitions it is desirable to support splicing parameter names irrespective of name consistency. With the proposed solution we can utilize `variable_of` to get splicing without any additional changes to [P2996R12].

```
void fun(int const a);
void fun(int const b) {
    [:expand(parameters_of(^fun)):] >> [&]<auto e> {
        constexpr auto param_variable = variable_of(e);
        cout << identifier_of(param_variable) << "=" << [:param_variable:] << ", ";
    };
}
```

11 Explicit Template Specializations

We recognize that primary template and explicit template specializations are special case. Following the discussion in the EWG, we propose adopting a conservative approach by enforcing naming consistency, as outlined in the previously described rules, between a templated function and any of its explicit specializations.

```
template <typename T>
void fun1(T x);

void fun1<float>(float x) {}
static_assert(has_identifier(parameters_of(^^fun1<float>)[0]));

template <typename T>
void fun2(T x);

void fun2<float>(float y) {}
static_assert(!has_identifier(parameters_of(^^fun2<float>)[0]));

template <typename T>
void fun3(T);

void fun3<float>(float y) {}
static_assert(has_identifier(parameters_of(^^fun3<float>)[0]));

template<class ...T>
void fun4(T... x);

template<> void fun4(int a, int b);

static_assert(!has_identifier(parameters_of(^^fun4<char>)[0]));
static_assert(!has_identifier(parameters_of(^^fun4<int, int>)[0]));
```

12 Explicit Template Instantiations

We recognize that primary template and its explicit instantations need to be goverened by a separate set of rules. Following the discussion in the EWG, we propose that explicit template instantiations are ignored for the purposes of parameter identifier reflection.

```
template <typename T>
void fun(T x);

template void fun<int>(int z);
static_assert(has_identifier(parameters_of(^^fun<int>)[0]));
static_assert(identifier_of(parameters_of(^^fun<int>)[0]) == "x"sv);
```

13 Default Arguments

For completeness sake, we are mentioning default arguments for parameters. In our research, we have encountered the need to know whether a parameter has a default argument, but not necessarily what the argument is. Therefore, we only propose to add the `has_default_argument` as it was already in [P1240R2].

14 Note on ODR Violations

Reflection on function parameters depends on which function declarations are reachable in the reflection context. While the proposed "Enforced Consistency" approach reduces the likelihood of ODR violations it does not eliminate it completely. It should be noted, that this problem does not pertain only to function parameters reflection, but also to reflecting namespaces or incomplete types. Therefore we defer to the authors of [P2996R12] for a more generic solution of the problem.

15 Proposed Metafunctions

15.1 parameters_of

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

Given a reflection `r` that designates a function, return a vector of the reflections of the explicit parameters. Results of this metafunction are not spliceable unless used in conjunction with variable_of (see later) inside the function definition.

15.2 identifier_of

```
namespace meta {
  consteval string_view identifier_of(info r);
  consteval u8string_view u8identifier_of(info r);
}
```

Given a reflection `r` that designates a function parameter, for which `has_identifier` evaluates to `true`, return a `string_view` or a `u8string_view` holding the name of that parameter. If `r` designates a function parameter, for which `has_identifier` evaluates to `false` then `identifier_of` and `u8identifier_of` are not constant expressions.

15.3 variable_of

```
namespace meta {
  consteval info variable_of(info r);
}
```

Given a reflection `r` that designates a function parameter, return the reflection of that parameter, as if obtained by reflecting its identifier.

15.4 return_type_of

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

Given a reflection `r` that designates a function or a function type, return a reflection of the return type of that function.

15.5 type_of

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

Given a reflection `r` that designates a function parameter, return the reflection of the type of the parameter after applying type adjustment.

15.6 has_identifier

```
namespace meta {
  consteval bool has_identifier(info r);
}
```

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

15.7 has_ellipsis_parameter

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

Given a reflection `r` that represents a function, return `true` if that function's parameter clause terminates with ellipsis. Otherwise `false`.

15.8 has_default_argument

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

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

15.9 is_explicit_object_parameter

```
namespace meta {
  consteval bool is_explicit_object_parameter(info r);
}
```

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

15.10 is_function_parameter

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

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

16 Proposed Wording

Wording is relative to [P2996R12]

16.1 Language

16.1.1 [basic.fundamental] Fundamental types

Add new bullet point x.6 to paragraph 16 of 6.8.2 [basic.fundamental].

16 The types denoted by *cv* `std::nullptr_t` are distinct types. [...]

x A value of type std::meta::info is called a *reflection*. There exists a unique *null reflection*; every other reflection is a representation of

(x.1) — a value of structural type (13.2 [temp.param]),

(x.2) — an object with static storage duration (6.7.6 [basic.stc]),

(x.3) — a variable (6.1 [basic.pre]),

(x.4) — a structured binding (9.7 [dcl.struct.bind]),

(x.5) — a function (9.3.4.6 [dcl.fct]),

(x.6) — a function parameter (9.3.4.6 [dcl.fct]),

(x.7) — an enumerator (9.8.1 [dcl.enum]),

(x.8) — a type alias (9.2.4 [dcl.typedef]),

(x.9) — a type (6.8 [basic.types]),

(x.10) — a class member (11.4 [class.mem]),

(x.11) — an unnamed bit-field (11.4.10 [class.bit]),

(x.12) — a primary class template (13.1 [temp.pre]),

(x.13) — a function template (13.1 [temp.pre]),

(x.14) — a primary variable template (13.1 [temp.pre]),

(x.15) — an alias template (13.7.8 [temp.alias]),

(x.16) — a concept (13.7.9 [temp.concept]),

(x.17) — a namespace alias (9.9.3 [namespace.alias]),

(x.18) — a namespace (9.9.1 [basic.namespace.general]),

(x.19) — a direct base class relationship (11.7.1 [class.derived.general]) or

(x.20) — a data member description (11.4.1 [class.mem.general])

A reflection is said to *represent* the corresponding construct.

16.2 Library

16.2.1 [meta.reflection.synop] Header <meta> synopsis

Modify [meta.reflection.synop] as follows:

```
namespace meta {
  [...]
  consteval bool is_copy_assignment(info r);
  consteval bool is_move_assignment(info r);
  consteval bool is_destructor(info r);

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

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


  [...]
  consteval info template_of(info r);
  consteval vector<info> template_arguments_of(info r);
  consteval vector<info> parameters_of(info r);
  consteval info variable_of(info r);
  consteval info return_type_of(info r);
}
```

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

- Add new conditions 1.6, 1.7 and renumber the subsequent conditions accordingly.
- Add new condition 4.3 and renumber the subsequent conditions accordingly.

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

1 *Returns:*

(1.1) — If `r` represents an entity that has a typedef name for linkage purposes (9.2.4 [dcl.typedef]), then `true`.

(1.2) — Otherwise, if `r` represents an unnamed entity, then `false`.

(1.3) — Otherwise, if `r` represents a class type, then `!has_template_arguments(r)`.

(1.4) — Otherwise, if `r` represents a function, then `true` if `!has_template_arguments(r)` and the function is not a constructor, destructor, operator function, or conversion function. Otherwise, `false`.

(1.5) — Otherwise, if `r` represents a template, then `true` if `r` does not represent a constructor template, operator function template, or conversion function template. Otherwise, `false`.

(1.6) — Otherwise, if `r` represents the i-th parameter of a function F that is an (implicit or explicit) specialization of a templated function T and the i-th parameter of the instantiated declaration of T whose template arguments are those of F would be instantiated from a pack, then `false`.

(1.7) — Otherwise, if `r` represents the parameter P of a function F, then let S be the set of declarations, ignoring any explicit instantiations, that precede some point in the evaluation context and that declare either F or a templated function of which F is a specialization; `true` if

(1.7.1) — there is a declaration D in S that introduces a name N for either P or the parameter corresponding to P in the templated function that D declares and

(1.7.2) — no declaration in S does so using any name other than N.

Otherwise, `false`.

[*Example:*

```
void fun(int);
constexpr std::meta::info r = parameters_of(^^fun)[0];
static_assert(!has_identifier(r));

void fun(int x);
static_assert(has_identifier(r));

void fun(int x);
static_assert(has_identifier(r));
```

```
void poison() {
    void fun(int y);
}
static_assert(!has_identifier(r));
```

— end example]

(1.8) — Otherwise, if `r` represents a variable, then `false` if the declaration of that variable was instantiated from a function parameter pack. Otherwise, `!has_template_arguments(r)`.

(1.9) — Otherwise, if `r` represents a structured binding, then `false` if the declaration of that structured binding was instantiated from a structured binding pack. Otherwise, `true`.

(1.10) — Otherwise, if `r` represents a type alias, then `!has_template_arguments(r)`.

(1.11) — Otherwise, if `r` represents an enumerator, non-static data member, namespace, or namespace alias, then `true`.

(1.12) — Otherwise, if `r` represents a direct base class relationship, then `has_identifier(type_of(r))`.

(1.13) — Otherwise, `r` represents a data member description (T, N, A, W, NUA) (11.4.1 [class.mem.general]); true if N is not $\perp$. Otherwise, `false`.

```
consteval string_view identifier_of(info r);
consteval u8string_view u8identifier_of(info r);
```

2 Let E be UTF-8 if returning a `u8string_view`, and otherwise the ordinary literal encoding.

3 *Constant When:* `has_identifier(r)` is `true` and the identifier that would be returned (see below) is representable by E.

4 *Returns:* An NTMBS, encoded with E, determined as follows:

(4.1) — If `r` represents an entity with a typedef name for linkage purposes, then that name.

(4.2) — Otherwise, if `r` represents a literal operator or literal operator template, then the *ud-suffix* of the operator or operator template.

(4.3) — Otherwise, if `r` represents the parameter P of a function F, then let S be the set of declarations, ignoring any explicit instantiations, that precede some point in the evaluation context and that declare either F or a templated function of which F is a specialization; the name that was introduced by a declaration in S for the parameter corresponding to P.

(4.4) — Otherwise, if `r` represents an entity, then the identifier introduced by the declaration of that entity.

(4.5) — Otherwise, if `r` represents a direct base class relationship, then `identifier_of(type_of(r))` or `u8identifier_of(type_of(r))`, respectively.

(4.6) — Otherwise, `r` represents a data member description (T, N, A, W, NUA) (11.4.1 [class.mem.general]); a `string_view` or `u8string_view`, respectively, containing the identifier N.

16.2.3 [meta.reflection.queries] Reflection queries

— Modify paragraph 1.
— Add a new condition 3.1 and renumber the subsequent conditions accordingly.
— Insert new paragraphs 53 through 63 after the existing content.

```
consteval bool has-type(info r); // exposition only
```

1 *Returns*: `true` if `r` represents a value, object, variable, function whose type does not contain an undeduced placeholder type and that is not a constructor or destructor, enumerator, non-static data member, unnamed bit-field, direct base class relationship, ~~or~~data member description, or function parameter. Otherwise, false.

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

2 *Constant When*: *has-type*(`r`) is `true`.

3 *Returns*:

(3.1) — If `r` represents the *i*-th parameter of a function *F*, then the *i*-th type in the parameter-type-list of *F* (9.3.4.6 [dcl.fct]).

(3.2) — ~~If~~Otherwise, if `r` represents a value, object, variable, function, non-static data member, or bit-field, then the type of what is represented by `r`.

(3.3) — Otherwise, if `r` represents an enumerator *N* of an enumeration *E*, then:

(3.3.1) — If *E* is defined by a declaration *D* that is reachable from a point *P* in the evaluation context and *P* does not occur within an *enum-specifier* of *D*, then a reflection of *E*.

(3.3.2) — Otherwise, a reflection of the type of *N* prior to the closing brace of the *enum-specifier* as specified in 9.8.1 [dcl.enum].

(3.4) — Otherwise, if `r` represents a direct base class relationship, then a reflection of the type of the direct base class.

(3.5) — Otherwise, for a data member description (*T, N, A, W, NUA*) (11.4.1 [class.mem.general]), a reflection of the type *T*.

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

53 *Constant When:* `r` represents a function or a function type.

54 *Returns:*

(54.1) — If `r` represents a function *F*, then a `vector` containing reflections of the parameters of *F*, in the order they appear in a declaration of *F*.

(54.2) — Otherwise, `r` represents a function type *T*; a `vector` containing reflections of the types in the parameter-type-list (9.3.4.6 [dcl.fct]) of *T*, in the order they appear in the parameter-type-list.

```
consteval info variable_of(info r);
```

55 *Constant When:*

(55.1) — `r` represents a parameter of a function *F* and

(55.2) — there is a point *P* in the evaluation context for which the innermost non-block scope enclosing *P* is the function parameter scope (6.4.4 [basic.scope.param]) associated with *F*.

56 *Returns:* The reflection of the parameter variable corresponding to `r`.

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

57 *Constant When:* Either `r` represents a function and *has-type*(`r`) is `true` or `r` represents a function type.

58 *Returns:* The reflection of the return type of the function or function type represented by `r`.

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

59 *Returns:* `true` if `r` represents a function or function type that has an ellipsis in its parameter-type-list (9.3.4.6 [dcl.fct]). Otherwise, `false`.

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

60 *Returns*: If `r` represents a parameter *P* of a function *F*, then:

(60.1) — If F is a specialization of a templated function T, then `true` if there exists a declaration D of T that precedes some point in the evaluation context and D specifies a default argument for the parameter of T corresponding to P. Otherwise, `false`.

(60.2) — Otherwise, if there exists a declaration D of F that precedes some point in the evaluation context and D specifies a default argument for P, then `true`.

Otherwise, `false`.

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

61 *Returns:* `true` if `r` represents a function parameter that is an explicit object parameter (9.3.4.6 [dcl.fct]). Otherwise, `false`.

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

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

17 Implementation Experience

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

- `identifier_of` https://godbolt.org/z/cvW4e1cj1
- `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
- `return_type_of` https://godbolt.org/z/c35vo7Erh

18 Polls

18.1 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

18.2 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

18.3 P3096R4: LEWG, November 2024, WG21 meeting in Wrocław

Poll: We want type_of(parameters_of(^^f)[i]) to return the reflection of a type-adjusted parameter type

SF: 10 F: 8 N: 1 A: 1 SA: 0

Poll: The function described in P3096R4 called "parameter_variable" should be called:

parameter_variable: 7 variable_of: 18 get_parameter_variable: 0 get_parameter_of: 0

Poll: We want the "parameter_variable(parameters_of(^^f)[i])" (renamed into variable_of) - i.e. "variable_of(parameters_of(^^f)[i])" - to return the reflection of the ith function parameter as if it was reflected by its name (calling parameter_variable is only allowed in the definition).

Unanimous consent.

18.4 P3096R5: LEWG, January 21st 2025 Telecon

Poll: We would like to modify the behaviour of existing function "identifier_of" by modifying from being ill-formed for either non-existing names or inconsistent param names, to having "identifier_of" function returning func params even if inconsistent through declarations

SF: 1 F: 2 N: 6 A: 4 SA: 3

Outcome: No consensus for a change

Poll: We encourage the addition of "any_identifier_of" returning names even if inconsistent or doesn't exist (to be explored in a follow-up paper).

SF: 3 F: 9 N: 1 A: 1 SA: 1 Outcome: Consensus in favor

Poll: Fix the wording (and approve by a CWG expert) and then approve the modification as proposed in P3096R5 for: identifier_of, u8identifier_of, type_of, has_identifier to support function parameters.

SF: 2 F: 12 N: 1 A: 0 SA: 1

Outcome: Consensus in favor

Poll: Fix the wording as above and approve the addition of the new functions to handle function param names: return_type_of, has_ellipsis_parameter, variable_of, has_default_argument, is_explicit_object_parameter, is_function_parameter (in addition to the previous poll modifying behaviour of already forwarded functions) and forward next revision to (P3096R6*) to LWG and CWG for C++26.

SF: 2 F: 12 N: 0 A: 2 SA: 1

Outcome: Weak consensus in favor

18.5 D3096R7: EWG, February 2025, WG21 meeting in Hagenberg

Forward to CWG for inclusion in C++26.

SF: 9 F: 10 N: 7 A: 3 SA: 0

Outcome: Consensus in favor.

18.6 P3096R10: LWG, June 2025, WG21 meeting in Sofia

Put P3096R10 (pending review by JW and CT) pending CWG approval to C++26?

F: 11 N: 0 A: 0

18.7 P3096R11: LEWG, June 2025, WG21 meeting in Sofia

Approve the changes in "P3096R11: Function Parameter Reflection in Reflection for C++26" for C++26.

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

18.8 P3096R11: EWG, June 2025, WG21 meeting in Sofia

EWG saw this paper and answer CWG's question:

EWG agrees with the breakout group's decision "if names of parameters don't match between specialization and primary template, user gets no name" and returns this paper to CWG.

SF: 23 F: 14 N: 6 A: 0 SA: 0

EWG confirms that explicit instantiations are ignored for the purposes of has_identifier/get_identifier reflection of parameter:

SF: 15 F: 20 N: 9 A: 0 SA: 0

19 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

[P2996R11] Barry Revzin, Wyatt Childers, Peter Dimov, Andrew Sutton, Faisal Vali, Daveed Vandevoorde, Dan Katz. 2025-04-16. Reflection for C++26.
https://wg21.link/p2996r11

[P2996R12] Barry Revzin, Wyatt Childers, Peter Dimov, Andrew Sutton, Faisal Vali, Daveed Vandevoorde, Dan Katz. 2025-05-17. Reflection for C++26.
https://wg21.link/p2996r12

[P2996R5] Barry Revzin, Wyatt Childers, Peter Dimov, Andrew Sutton, Faisal Vali, Daveed Vandevoorde, Dan Katz. 2024-08-14. Reflection for C++26.
https://wg21.link/p2996r5

[P2996R7] Barry Revzin, Wyatt Childers, Peter Dimov, Andrew Sutton, Faisal Vali, Daveed Vandevoorde, Dan Katz. 2024-10-13. Reflection for C++26.
https://wg21.link/p2996r7