Function Parameter Reflection in Reflection for C++26

Document #: P3096R4
Date: 2024/11/21
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 [P2996R7].

2 Revisions

2.1 R4

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

2.2 R3

— improved proposed wording
— aligned with [P2996R5]

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

2.4 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

2.5 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 [P2996R7].

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/xP55oj416 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/ncnYdW7MK 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 `identifier_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. 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(...identifier_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 << 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 [P2996R7] - 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
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 values 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 [P2996R7] 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 `parameter_variable` 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(parameter_variable(parameters_of(^^fun)[0]) == ^^c);
  static_assert(is_const(
              parameter_variable(parameters_of(^^fun)[0])));
  static_assert(is_const_v<
              [: parameter_variable(parameters_of(^^fun)[0]) :]);
}
```

9 Reflecting on Parameter Names

9.1 Splicing in Function Definition

We recognize that within function definitions it is desirable to support splicing parameter names irrespective of name consistency

```
void fun(int const a);
void fun(int const b) {
    [:expand(parameters_of(^func)):] >> [&]<auto e> {
```

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

9.2 Proposal

We propose to extend the `identifier_of` metafunction specified in [P2996R7] to return the name of a function parameter if all reachable function declarations have the same name for that parameter or that parameter is unnamed.

```
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 `parameter_variable` 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(parameter_variable(parameters_of(^^fun)[0]) == ^^c);
  assert(identifier_of(parameter_variable(parameters_of(^^fun)[0])) == "c"sv);
}
```

10 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].

11 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 [P2996R7] for a more generic solution of the problem.

12 Proposed Metafunctions

12.1 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.

12.2 has_identifier

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

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`.

12.3 identifier_of

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

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.

12.4 parameter_variable

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

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

12.5 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`.

12.6 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`.

12.7 is_explicit_object_parameter

```
namespace meta {
  consteval auto is_explicit_object_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`.

12.8 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`.

12.9 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 of the declared return type of that function.

12.10 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 type adjustment.

13 Proposed Wording

Wording is relative to [P2996R5]

13.1 library

13.1.1 [meta.synop] Header `<meta>` synopsis

Modify the future 21.4 subsection in 21 [meta].

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

```
  consteval auto has_identifier(info r) -> bool;
  consteval auto identifier_of(info r) -> string_view;
  consteval auto u8identifier_of(info r) -> u8string_view;
  consteval auto parameter_variable(info r) -> info;
  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;
}
```

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

```
consteval auto identifier_of(info r) -> string_view;
consteval auto u8identifier_of(info r) -> u8string_view;
```

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

2 *Constant When:* If `r` represents a function, then when the function is not a constructor, destructor, operator function, or conversion function. Otherwise, if `r` is a function template, then when the function template is not a constructor template, a conversion function template, or an operator function template. Otherwise, if `r` represents a variable, an entity that is not a function or function template, or an alias of a type or namespace, then when the declaration of what is represented by `r` introduces an identifier representable by *E*. Otherwise, if `r` represents a base class specifier for which the base class is a named type, then when the name of that type is an identifier representable by *E*. Otherwise, when `r` represents a description of the declaration of a non-static data member, and the declaration of any data member having the properties represented by `r` would introduce an identifier representable by *E*. Otherwise, when `r` represents a function parameter that is not declared with different names in all reachable declarations.

3 *Returns:*

- (3.1) — If `r` represents a literal operator or literal operator template, then the ud-suffix of the operator or operator template.
- (3.2) — Otherwise, if `r` represents a variable, entity, or alias of a type or namespace, then the identifier introduced by the declaration of what is represented by `r`.
- (3.3) — Otherwise, if `r` represents a base class specifier, then the identifier introduced by the declaration of the type of the base class.
- (3.4) — Otherwise, if `r` represents a description of the declaration of a non-static data member, then the identifier that would be introduced by the declaration of a data member having the properties represented by `r`.
- (3.5) — Otherwise, if `r` represents a function parameter, then the name of that parameter. If that parameter is unnamed in all reachable declarations then an empty string.

```
consteval auto parameter_variable(info r) -> info;
```

8 *Constant When:* `r` denotes a named function parameter, and `parameter_variable` is invoked within the function-body of the function to which that parameter belongs. *Returns:* the reflection of the function parameter a represented by `r`, as if obtained by reflecting its identifier (`^^a`).

13.1.3 [meta.reflection.queries] Reflection queries

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

33 *Constant When:* `r` represents a value, object, variable, function that is not a constructor or destructor, enumerator, non-static data member, bit-field, base class specifier, ~~or~~ description of the declaration of a non-static data member, or a function parameter.

34 *Returns*: If `r` represents an entity or variable for which every declaration specifies its type using the same *typedef-name*, a base class specifier whose base class is specified using a *typedef-name*, or a description of the declaration of a non-static data member whose type is specified using a *typedef-name*, then a reflection of the *typedef-name*. Otherwise, if `r` represents an entity or variable, then the type of what is represented by `r`. Otherwise, if `r` represents a base class specifier, then the type of the base class. Otherwise, the type of any data member declared with the properties represented by `r`. If `r` represents a function parameter whose type is an “array of T”, then the reflection of a non-volatile, non-const qualified “pointer to T” type. If `r` represents a function parameter of any other type, then the reflection of the non-volatile, non-const-qualified type of that parameter.

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

46 *Constant When:* `r` represents a function that is not a function template.

47 *Returns:* The reflections of the parameters of the function represented by `r`.

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

48 *Returns:* `true` if `identifier_of(r)` is a core constant expression. Otherwise `false`.

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

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

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

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

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

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

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

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

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

53 *Constant When:* `r` represents a function that is not a constructor or a destructor.

Returns: If `r` represents a function for which every reachable declaration specifies its return type using the same *typedef-name*, then a reflection of the *typedef-name*. Otherwise the declared return type of `r`, when that type is a *typedef-name* then the type denoted by the *typedef-name*.

14 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
- `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
- `return_type_of` https://godbolt.org/z/c35vo7Erh

15 Polls

15.1 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

15.2 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

16 Acknowledgments

We'd like to thank Daveed Vandevoorde, Sergei Murzin, Joshua Berne, 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.

17 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

[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