Argument type deduction for non-trailing parameter packs

Document #:	P2347R2
Date:	2021-10-13
Programming Language C++	
Audience:	EWG

Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>
	Bruno Manganelli <bruno.manga95@gmail.com>

Abstract

We propose that, during template argument deduction, a single non-trailing parameters pack be deduced solely based on the arity of the list of arguments. This paper proposes the mechanism for function calls, but a generalization is explored.

Revisions

Revision 2

Exploration and implementation experience of argument deduction in more contexts (class, function address, aliases, partial specialization).

Revision 1

Fix many typos in the wording, remove incorrect wording examples.

Status and target vehicle

The proposal has been implemented, and we propose some wording, although that wording is only for function calls, and is insufficient. As requested by EWG, the paper also explores non-trailing pack deduction for type deduction outside of function calls, and this has been implemented too, but there are some open questions, some risks (the authors are not sure they covered all the bases), and no wording.

We can either:

- Target the whole proposal at C++26
- Try to ship type deduction for function calls in C++23, as this covers the original `source_location` use case and generalize later.

This is a scenario where incremental improvements are possible.

Motivation

`source_location`

Our primary motivation for this change is to improve the usability of `source_location`. Many loggers, especially those using fmt, such as `spdlog` offer a log function of the following form:

```
void log(string_view formatString, auto&&...args);
```

Which can then be called with arbitrary arguments: `log("Hello !", "world");`

Naturally, we would like to extend this function to support source location, and offer a more useful logging framework:

```
void log(string_view formatString, auto&&...args, source_location loc = source_location::current());
```

Unfortunately, this is not possible because non-trailing packs cannot be deduced! Folks on Stackoverflow have found several clever workarounds, all of which involves using extra types or templates. So, even if the use case can be somewhat covered, it relies on rather arcane solutions that are worse for diagnostics, compile-time, etc.

Accessing the last argument of a pack

It is sometimes useful to handle the last parameter differently. The following example is taken from P0478R0 [1]:

```
template <class... Args, class Last>
void signal(Args... args, Last last) {
    // callback expects 5 arguments, and we only want to pass it the first 5
    if constexpr(sizeof... (Args) > 5) {
        return signal(args...);
    } else if constexpr (sizeof... (Args) == 4) {
        callback(args..., last);
    } else {
        callback(args...);
    }
}
```

Or consider that function which prints its arguments:

```
void print(auto&&... args, auto && last) {
    if constexpr(sizeof...(args) > 0)
        ((std::cout << args << ", "),...);
    std::cout << last << "\n";
}
```

This is currently rather difficult.

Or a usage of `apply` that handles the last argument differently:

```
std::apply([](auto&&..., auto && last) {
```

```
    assert(last == 3);
}, std::tuple{1, 2, 3});
```

A `apply_last` function can be written, albeit it's a bit cumbersome.

```
template <class F, class Tuple>
constexpr decltype(auto) apply_last(F &&f, const Tuple &t) {
    return [&]<auto... I>(std::index_sequence<I...>) {
        return f(std::get<std::tuple_size_v<std::remove_reference_t<Tuple>> - 1>(t),
        std::get<I>(t)...);
    }(std::make_index_sequence<std::tuple_size_v<std::remove_cvref_t<Tuple>> -1>{});
}
```

Consistent interfaces with variadic arguments

We might consider providing N ranges overloads to `std::transform`, `std::merge` and similar algorithms, such that they are consistent with the order of parameters of existing 1 and 2 ranges overloads.

We would also argue that `visit` would be more intuitive if the variants were the first parameters.

Lifting the limitations on where a parameter pack can appear gives more flexibility in API design and usage.

keys-value interfaces

Another great use, suggested by the author of sol2, is an interface that takes a number of keys (corresponding to a path or a nested addressing scheme of some sort) and a value. Users' expectations and logic would dictate that the value would be the last parameter. This proposal, therefore, allows for less surprising APIs.

```
// given a lua table (or json doccument, or something of that nature), set("a", "b", "c", 42)
// table["a"]["b"]["c"] becomes 42
void set_value_in_lua_table(auto&& table, auto... keys, auto value);
```

Design

We propose that if there is one (and only one) parameter pack in a function, the arity of that parameter pack, when deduced, is the number of not yet deduced function arguments, minus the number of non-defaulted parameters following the pack.

The general idea is to deduce a single pack and to deduce the size of that pack such that once expanded, the argument list matches the size of the parameter list, excluding any defaulted parameter.

```
void f(auto a, auto...b, auto c, auto d);
void g(auto a, auto...b, auto c, int d = 0);
void h(auto a, auto...b, int c = 0);
```

```
f(0, 0, 0, 0);      // size of b is deduced to be 1
f(0, 0, 0, 0, 0);   // size of b is deduced to be 2
f(0, 0, 0);         // size of b is deduced to be 0

g(0, 0);           // size of b is deduced to be 0
g(0, 0, 0, 0);     // size of b is deduced to be 2

h(0, 0);           // size of b is deduced to be 1
h(0, 0, 0);        // size of b is deduced to be 2
```

Unlike P0478R0 [1], we do not propose that the compiler should try to deduce a valid overload with or without default parameter or apply a more clever logic. This proposal is based solely on the arity of the arguments. This is why we consider this paper lifts a restriction rather than introducing a new feature. We do not propose any changes to overload resolution nor the ordering of function templates.

As such, a limitation of this proposal is that if a parameter pack is immediately followed by a parameter P with a default value, it is not possible for the caller to provide a value for P.

```
void f(auto...a, int c = 42);
f()     // a is empty, c == 42
f(1)    // a is of size 1,  c == 42
f(1, 2) // a is of size 2,  c == 42
```

We found that trying to be clever here is not likely to be worth it:

- Generating automatically extra overloads for each defaulted parameter has a cost in compile times.
- It would blur the lines between template argument deduction and overload resolution.

If one really needs a defaulted argument immediately following a pack, it is always possible to manually craft an overloads set that would allow a parameter to be both provided and defaulted, for example:

```
template <typename... T>
void f(T&&... args, source_location loc = {})
requires (!(std::same_as<T, source_location>||...));

void f(auto&&... args, source_location loc);
```

Interesting and breaking changes

Non-trailing packs in function are no longer non-deduced

```
void ambiguous(auto..., auto);    // #1
void ambiguous(auto, auto...);    // #2
void ambiguous(auto..., auto...); // #3
```

```
ambiguous(1, 2);
```

In C++20, #1 and #3 have undeduced packs, so the only viable candidate after overload resolution is #2. With this proposal, all arguments of both #1 and #2 can be deduced. None of the arguments of #3 can be deduced (multiple packs are never deduced), and after overload resolution, both #1 and #2 are viable overloads, and the call is ambiguous. A program that was well-formed in C++20 would be ill-formed with this proposal.

Note that in that scenario, it is not possible to resolve the ambiguity with a cast, for example. This is not a new problem; functions with undeduced packs cannot be explicitly named when there are ambiguities in C++20.

Multi levels templates

The implementation in clang revealed this interesting case.

```
template<typename...T>
struct S {
    template<typename...U>
    void g(U &&...u, T &&...t) {}
};

void test_nested_packs() {
    S<int>().g(0, 1);  // #1
}
```

After instantiation of `S`, `g` is

```
template<typename...U>
void g(U &&...u, int);
```

In C++20, `U...` is an undeduced context, so #1 is ill-formed. With is proposal, `U` can be deduced to `int` and the program is well-formed.

Deduction in other contexts: Implementation and use cases

In addition to function calls, type deduction (where a pack can appear) can occur when

- Instantiating template classes
- Taking the address of a function template
- Partial ordering

EWG asked that these things are explored. This section is a report on that exploration. **Note that type deduction for function arguments could be standardized independently of everything else and would provide benefits for users of `source_location` in the C++23 time frame.** The exploration shows that this can be expanded later with a consistent design.

Type deduction during class instantiation

A motivating use case is a meta-type that expand to the last argument of a template.

```
template <typename...>
struct mp_list{};

// Extracting the last element in a type list is made much easier
template<class L>
struct pop_back_impl{};
template<template<class...> class L, class... T, class Last>
struct pop_back_impl<L<T..., Last>> {
    using type = L<T...>;
};

template<class L>
using pop_back = typename pop_back_impl<L>::type;
```

This has been implemented in Compiler Explorer. Note that currently, non-trailing parameter packs in class template heads are ill-formed, whereas they are only non-deduced for function calls. Unlike function calls, this has, therefore, no risk of affecting existing code.

This works exactly like for function calls. Each time deduction happens, we compare a list of arguments to a list of parameters, and so we can deduce the size of a single pack from the arity of a single pack. There cannot be multiple packs for this to work Compiler Explorer. This is one of the reasons this paper does not replace P1858R2 [2].

The same logic would apply anywhere a type or an alias is instanciated, for example:

```
template<class..., class T>
using  Last = T;
static_assert(std::same_as<Last<int, int, float>, float>);
```

Function address

The address of a function template can be deduced in the same way as being only a special case of argument deduction, where the list of arguments is extracted from the type of the LHS declaration. [Compiler Explorer]

```
auto f(auto... a, int x = 0) {
    return(a +...) * x;
}
int g() {
    using F = int(int, int, int);
    F& func = f;

    return func(20, 1, 2);
}
```

Partial specialization

Partial specialization also follows regular type deduction rules [Compiler Explorer].

CTAD

Class Template Arguments Deduction also follows naturally from the same rule. Nothing specific had to be modified in Clang to make this work [Compiler Explorer]

CTAD for aggregate has not been explored.

Defaulted template parameters

There is a design question of whether to allow template heads of the form

```
template <typename...T, typename U = void>
```

It might be useful SFINAE tricks or to declare verbose types used subsequently in the declaration, but like non-trailing defaulted parameters, they could never be deduced and would always have their default value. The two design choices are

- Ill-formed (this is what the implementation is doing)
- Well-formed and always hold the default value.

Previous works

P0478R0 [1] was first presented in Issaquah and offered a more complicated approach to some of the problems presented here. Concerns were expressed mostly because P0478R0 [1] proposed to modify rules around overload resolution, which the current paper does not. It is, therefore, a lot simpler.

We also consider that `source_location` demands that this question be revisited.

State of the implementation

There was no major issue implementing what is covered in this paper in clang; although I didn't probably cover all scenarios, this is very much a prototype. I am hoping that the existence of that prototype might let us uncover scenarios that we might think about. There is probably nothing worth reporting on the implementation. It was probably a week worth of work, which tells more about how bad a compiler writer I am than it does about the actual amount of work required for this feature.

It was necessary for clang to keep track of the index of the template parameter each argument was expanded to. Some work went into properly handling explicit template arguments.

Alternatives and future evolutions

Generalized pack manipulation facilities

Several proposals, including P1858R2 [2] and P1306R1 [3], would make manipulating pack simpler, and we hope these papers progress. However, neither of these could address the `source_location` issue (which we realize is rather specific) and are not as elegant in some use cases.

Pack separators

Circle provides a syntax to denotes the end of a pack, which allows separating a pack from subsequent defaulted arguments, and also to support the deduction of multiple packs as described in the Circle documentation. We are not proposing a similar feature, but it is something that we could consider in the future in a backward-compatible manner.

Injecting multiple overloads for different combinations of defaulted parameters

This is discussed in a previous section, and it is a direction we rejected because of its cost and complexity. It is important to note that adopting such a feature in the future would be a breaking change in regard to this paper.

Implementation

Wording (For function templates ONLY!)

[temp.deduct] Template argument deduction

[temp.deduct.call] Deducing template arguments from a function call

//...

For a function parameter pack ~~that occurs at the end of the *parameter-declaration-list* ~~,the size of the pack `N` is deduced to be the number of arguments in the call, minus the number of arguments preceding `A`, and minus one for each function template parameter following `P` that has no default argument.

~~d~~ Deduction is performed for ~~each remaining~~ the next `N` argument~~s~~ of the call, taking the type `P` of the *declarator-id* of the function parameter pack as the corresponding function template parameter type. Each deduction deduces template arguments for subsequent positions in the template parameter packs expanded by the function parameter pack. When a function parameter pack appears in a non-deduced context, the type of that pack is never deduced.

[*Example:*

```
template<class ... Types> void f(Types& ...);
template<class T1, class ... Types> void g(T1, Types ...);
```

```
    template<class T1, class ... Types> void g1(Types ..., T1);

    void h(int x, float& y) {
        const int z = x;
        f(x, y, z);                  // Types deduced as int, float, const int
        g(x, y, z);                  // T1 deduced as int; Types deduced as float, int
        g1(x, y, z);                 // error: Types is not deduced
        g1<int, int, int>(x, y, z);  // OK, no deduction occurs
    }

template<class ... Types> void f(Types& ...);
template<class T1, class ... Types> void g(T1, Types ...);
template<class T1, class ... Types> void g1(Types ..., T1);
template<class ... Types> void g2(int, Types ..., int = 0);
template<class ... Types, class T2, class... OtherTypes> void g3(Types ..., T2, OtherTypes...);

void h(int x, float& y) {
    const int z = x;
    f(x, y, z);  // Types deduced as int, float, const int
    g(x, y, z);  // T1 deduced as int; Types deduced as float, int
    g1(x, y, z); // Types deduced as float, int, T1 deduced as int;
    g2(x, x);    // Types deduced as int;
    g2(x);       // sizeof...(Types) == 0;
    g3(x, y, z); // error: Types is not deduced, OtherTypes is not deduced
    g3<int, int, int>(x, y, z);   // OK, no deduction occurs
}
```

— end example]

[ ] Deducing template arguments during partial ordering [temp.deduct.partial]

Two sets of types are used to determine the partial ordering. For each of the templates involved there is the original function type and the transformed function type. [*Note:* The creation of the transformed type is described in [temp.func.order]. *— end note*] The deduction process uses the transformed type as the argument template and the original type of the other template as the parameter template. This process is done twice for each type involved in the partial ordering comparison: once using the transformed template-1 as the argument template and template-2 as the parameter template and again using the transformed template-2 as the argument template and template-1 as the parameter template.

The types used to determine the ordering depend on the context in which the partial ordering is done:

- In the context of a function call, the types used are those function parameter types for which the function call has arguments.
- In the context of a call to a conversion function, the return types of the conversion function templates are used.
- In other contexts the function template's function type is used.

Each type nominated above from the parameter template and the corresponding i^{th} type from the argument template are used as the types of `P` and `A`. Let `PCount` be the numbers of types of the parameter template, and `ACount` be the numbers of type of the argument template.

Before the partial ordering is done, certain transformations are performed on the types used for partial ordering:

- If `P` is a reference type, `P` is replaced by the type referred to.
- If `A` is a reference type, `A` is replaced by the type referred to.

If both `P` and `A` were reference types (before being replaced with the type referred to above), determine which of the two types (if any) is more cv-qualified than the other; otherwise the types are considered to be equally cv-qualified for partial ordering purposes. The result of this determination will be used below.

Remove any top-level cv-qualifiers:

- If `P` is a cv-qualified type, `P` is replaced by the cv-unqualified version of `P`.
- If `A` is a cv-qualified type, `A` is replaced by the cv-unqualified version of `A`.

Using the resulting types `P` and `A`, the deduction is then done as described in **??**. If `P` is a function parameter pack, the type `A` of ~~each remaining~~ The next $ACount - Pcount - i$ parameter types of the argument template is compared with the type `P` of the *declarator-id* of the function parameter pack. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by the function parameter pack. Similarly, if `A` was transformed from a function parameter pack, it is compared with ~~each remaining~~ The next $ACount - Pcount - i$ parameter types of the parameter template. If deduction succeeds for a given type, the type from the argument template is considered to be at least as specialized as the type from the parameter template. [*Example:*

```
template<class... Args>           void f(Args... args);         // #1
template<class T1, class... Args> void f(T1 a1, Args... args);  // #2
template<class T1, class T2>      void f(T1 a1, T2 a2);         // #3

f();               // calls #1
f(1, 2, 3);        // calls #2
f(1, 2);           // calls #3; non-variadic template #3 is more specialized
// than the variadic templates #1 and #2
```

— end example]

[*Example:*

```
template<class... Args>             void f(Args... args);        // #1
template<class T1, class... Args>   void f(T1 a1, Args... args); // #2
template<class T1, class T2>        void f(T1 a1, T2 a2);        // #3

f();               // calls #1
f(1, 2, 3);        // calls #2
```

```
f(1, 2);            // calls #3; non-variadic template #3 is more specialized
// than the variadic templates #1 and #2
```

— *end example*]

If, for a given type, the types are identical after the transformations above and both `P` and `A` were reference types (before being replaced with the type referred to above):

- if the type from the argument template was an lvalue reference and the type from the parameter template was not, the parameter type is not considered to be at least as specialized as the argument type; otherwise,
- if the type from the argument template is more cv-qualified than the type from the parameter template (as described above), the parameter type is not considered to be at least as specialized as the argument type.

Function template `F` is *at least as specialized as* function template `G` if, for each pair of types used to determine the ordering, the type from `F` is at least as specialized as the type from `G`. `F` is *more specialized than* `G` if `F` is at least as specialized as `G` and `G` is not at least as specialized as `F`.

If, after considering the above, function template `F` is at least as specialized as function template `G` and vice-versa, and if `G` has a ~~trailing~~ function parameter pack for which `F` does not have a corresponding parameter, and if `F` does not have a ~~trailing~~ function parameter pack, then `F` is more specialized than `G`.

In most cases, deduction fails if not all template parameters have values, but for partial ordering purposes a template parameter may remain without a value provided it is not used in the types being used for partial ordering. [*Note:* A template parameter used in a non-deduced context is considered used. — *end note*] [*Example:*

```
template <class T> T f(int);            // #1
template <class T, class U> T f(U);     // #2
void g() {
    f<int>(1);                            // calls #1
}
```

— *end example*]

[*Note:* Partial ordering of function templates containing template parameter packs is independent of the number of deduced arguments for those template parameter packs. — *end note*] [*Example:*

```
template<class ...> struct Tuple { };
template<class ... Types> void g(Tuple<Types ...>);                 // #1
template<class T1, class ... Types> void g(Tuple<T1, Types ...>);   // #2
template<class T1, class ... Types> void g(Tuple<T1, Types& ...>);  // #3

g(Tuple<>());                   // calls #1
g(Tuple<int, float>());         // calls #2
```

```
g(Tuple<int, float&>());        // calls #3
g(Tuple<int>());                // calls #3
```

— *end example*]

[ ] Deducing template arguments from a type [temp.deduct.type]

The non-deduced contexts are:

- The *nested-name-specifier* of a type that was specified using a *qualified-id*.
- The *expression* of a *decltype-specifier*.
- A non-type template argument or an array bound in which a subexpression references a template parameter.
- A template parameter used in the parameter type of a function parameter that has a default argument that is being used in the call for which argument deduction is being done.
- A function parameter for which the associated argument is an overload set, and one or more of the following apply:
  - more than one function matches the function parameter type (resulting in an ambiguous deduction), or
  - no function matches the function parameter type, or
  - the overload set supplied as an argument contains one or more function templates.
- A function parameter for which the associated argument is an initializer list but the parameter does not have a type for which deduction from an initializer list is specified. [*Example:*

```
template<class T> void g(T);
g({1,2,3});                 // error: no argument deduced for T
```

  — *end example*]
- A function parameter pack that ~~does not occur at the end of~~ is not the only parameter pack in the *parameter-declaration-list*.

// ...

Feature test macros

[Editor's note: Add a new macro in `[tab:cpp.predefined.ft]` : `__cpp_non_trailing_function_pack` set to the date of adoption] .

Acknowledgments

Thanks to Sy Brand and Michael Wong for their work on P0478 and encouragements. Tony Van Eerd and Ólafur Waage for proofreading this paper and offering their feedbacks, and Jens Maurer for reviewing the wording.

References

[1] Bruno Manganelli, Michael Wong, and Sy Brand. P0478R0: Template argument deduction for non-terminal function parameter packs. https://wg21.link/p0478r0, 10 2016.

[2] Barry Revzin. P1858R2: Generalized pack declaration and usage. https://wg21.link/p1858r2, 3 2020.

[3] Andrew Sutton, Sam Goodrick, and Daveed Vandevoorde. P1306R1: Expansion statements. https://wg21.link/p1306r1, 1 2019.

[N4885] Thomas Köppe *Working Draft, Standard for Programming Language C++* https://wg21.link/N4885