Argument type deduction for non-trailing parameter packs

Document #:	P2347R1
Date:	2021-07-15
Programming Language C++	
Audience:	EWG

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

Abstract

We propose that, during function template argument deduction, a single non-trailing parameters pack be deduced solely based on the arity of the list of arguments.

Revisions

Revision 1

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

Motivation

`source_location`

Our main 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.

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);
```

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.

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.

Non-trailing template parameters of class and alias template

Similar restrictions on the positioning of parameter packs exist for class templates and alias. We could lift these restrictions using the same heuristic - aka using the number of template arguments to deduce the size of a single non-trailing parameter pack.

```
template <typename... T, typename> // ill-formed
struct S;
```

This is not proposed in this paper.

Implementation

Both this paper and P0478R0 [1] have been implemented in clang with no difficulties. In particular, this proposal required a very small amount of work.

Wording

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

//...

Let N be the number of remaining non-defaulted function template parameters, and K be the number of remaining arguments of the call.

For a function parameter pack ~~that occurs at the end of the *parameter-declaration-list* ~~, deduction is performed for ~~each remaining~~ the next K-N arguments 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*]

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

Using the resulting types P and A, the deduction is then done as described in [temp.deduct.type].

Let N be the number of remaining non-defaulted function template parameters and K be the number of remaining arguments of the call.

If P is a function parameter pack, the type A of each ~~remaining~~ of the next K-N 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~~ of the next N-K parameter type 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*]

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