Partial application of concepts in template arguments

Document #:	P2970R0
Date:	2025-05-19
Programming Language C++	
Audience:	EWG

Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>
	Gašper Ažman <gasper.azman@gmail.com>

Abstract

Concept template parameters were introduced in C++26 by P2841R7 [3]. We propose to extend this feature with partially applied concepts. This feature was introduced in P2841R0 [1] and removed in R1 to reduce the scope of the initial paper, and because EWG initially found this part of the feature less compelling.

It is presented again here, with added wording and notes on implementation experience.

Motivation

The first argument of a concept is the entity for which we are checking satisfaction. Concepts can take additional arguments parameterizing the concept.

For example, `invocable<T, int>` checks whether an object of type `T` can be called with an `int` argument.

In specific contexts, a `type-constraint` can appear in a template head or before `auto` to constrain — or assert — a type. In that case, the compiler injects the type to which the concept is applied, rather than the user specifying this type.

For example, the following two declarations are equivalent:

```
template<invocable<int> T>
void f();

template<typename T>
void f() requires invocable<T, int>;
```

Such shorthand is helpful because many concepts are binary or n-ary. In the `concepts` header, half of the concepts are not unary.

When passing concepts as template arguments, this becomes quickly apparent. Let us consider a `range_of` concept constraining the range's value type with a concept template-parameter:

```
template <typename R, template <typename> concept C>
concept range_of = std::ranges::range<R> && C<std::ranges::range_value_t<R>>;
```

We can then have an algorithm taking a range of `integral`:

```
auto median(range_of<std::integral> auto&&);
```

We may want to be more specific. Can we accept a range of only `int`s? Given that our `range_of` expects a concept (it should really accept a universal template parameter as described in P2989R2 [2]!), perhaps we could use `same_as`, but `same_as` requires arguments. Can we make something like that work?

```
auto f(range_of<same_as<int>> auto&&);
```

We can rewrite our `range_of` concept to take an extra argument pack forwarded to the concept parameter:

```
template <typename R, template <typename...> concept C,  typename... Args>
concept range_of = std::ranges::range<R> && C<std::ranges::range_value_t<R>, Args...>;
void f(range_of<std::same_as, int> auto &&);
```

This interface is not especially clear and, crucially, doesn't compose. For example, a range of `regular` types also convertible to `int` is not expressible.

One workaround would be to lift the concept and its argument into a type:

```
template <template <typename...> concept C, typename... Args>
struct packed_concept {
    template <typename T>
    static constexpr bool apply  = C<T, Args...>;
};

template <typename R, typename... PackedConcept>
concept range_of = std::ranges::range<R>
                   && (PackedConcept::template apply<std::ranges::range_value_t<R>> && ...);
void f(range_of<packed_concept<std::convertible_to, int>,
                packed_concept<std::regular>> auto &&);
```

This solution works, though it is not particularly usable or readable and loses all hope of subsumption. Concepts cannot be class members for a very good reason: Doing so would fundamentally break subsumption. So `apply` is a `bool` variable, an atomic constraint.

Can we still support making something similar in the language? Could we make something like this work?

```
void f(range_of<concept convertible_to<int>> auto)
```

This is the kind of solution we are proposing.

We can indeed create a new kind of template argument that carries additional arguments that are injected when the corresponding concept template-parameter is specialized:

```
// Given this declaration,
template <typename T, template <typename> concept C>
constexpr bool b = C<T>;
```

```
// this specialization
b<double, concept std::convertible_to<int>>;

// is rewritten by injecting the arguments passed to the concept argument.
template<>
constexpr bool b<double, concept convertible_to<int>> = convertible_to<double, int>;
```

Notice that in this example, `C` is a concept template-parameter that accepts exactly one argument, even though `convertible_to` needs two. The other arguments will be filled in automatically.

In effect, it's as if we created a new concept

```
template <typename T>
concept convertible_to_int = convertible_to<T, int>;
```

but did so inline, directly in the template argument. This feature, which we call *partial application*, expresses intent very clearly and works with subsumption.

Syntax of partially applied concepts

Partially applied concepts can appear as template arguments matching a concept template-parameter (and nowhere else). You might have noticed the `concept` keyword prefixing template arguments in previous examples and wonder why `range_of<concept convertible_to<int>>` rather than simply `range_of<convertible_to<int>>`?

Of course, an undeniable parallel exists with the syntax of concept declaration, which is nice, but the keyword avoids ambiguity.

If the concept accepts a variable number of arguments (because it has defaulted or variadic parameters), whether `ConceptName<Args>` is a partial application or a complete specialization (which is then a boolean expression) can be ambiguous.

This situation has not been an issue with *type-constraint*s because they appear only in a context where the first parameter is always injected.

However, the ambiguity between a partial concept application and a boolean expression can appear in a couple of cases.

1. In the presence of universal template parameters: Should a given argument matching a universal template parameter be considered a `bool` or a concept template-parameter?
2. In an overload set, the concept could be matched to both a `bool` and a concept, as in this example courtesy of Barry Revzin:

   ```
   template <bool B> void f();
   template <template <typename> concept C> void f();
   f<invocable<int>>(); // boolean or partially applied concept?
   ```

We explored various solutions to this ambiguity, including always treating something that *could* be a concept as a concept and forcing an explicit cast to bool to force a boolean expression or making the concept keyword optional when the concept has a fixed number of arguments.

Both approaches suffer the same issue: They could change the meaning of code when the concept is modified by adding defaulted or variadic parameters and could also break pre-C++26 code, so they appear to be nonideal solutions.

We also refrained from any solution that would make the nature of template arguments somehow deduced from the corresponding parameter during overload resolution, for this would be madness and would not really solve the question for universal template parameters.

Ultimately, the `concept` keyword is a great way to show intent and mirrors concept declaration. The following syntax would be valid:

```
some_template_name <
    std::regular,  /// can pass the name directly
    concept regular<>, // no reason this should not work
    invocable<int>, // That is a bool; it will get diagnosed.
                    // If no overload or specialization, expect a bool.
    concept invocable<int> // ok
>;
```

Can this feature be generalized to other kind of template template-parameters?

A question that arises when considering this partial application of concepts is whether it generalizes to other kinds of template parameters.

Concepts are unique in that the first parameter has a special meaning. That we could pass the first parameter at a different time than all other parameters does make a lot of sense for other kinds of template names. After all, `type-constraint`s exist for this purpose.

To generalize partial application to other template template-parameters, we would need the ability to provide — or not — arbitrary parameters; we could imagine, for example, being able to write some kind of code in which some but not all template arguments are provided.

```
Foo<std::map<?, ? , my_comparator>>;
```

However, this solution is more complicated than what we need for concepts, and its usefulness is debatable. How arguments and parameters would be matched is less clear. Besides, unlike concepts, aliases and variable templates can be created at class scope.

Therefore, the motivation for partially applied concepts does not necessarily generalize terribly well. We did consider whether this sort of placeholder syntax would be a good fit for concepts in isolation, but we think `concept` *`type-constraint`* is more consistent, less novel, and therefore more teachable than a placeholder syntax.

Should we support packs of partial concepts

Arguably, one could imagine a scenario in which a pack of partial concepts could be constructed:

```
template <typename T, template <typename> concept... Concepts>
concept AllOf  = (Concepts<T> && ...);
```

```
template <typename T, template <typename...> concept... Concepts>
concept Foo = AllOf<T, concept Concepts<double>...>;
```

But imagining a scenario in which this construction would be useful is difficult; i.e., when do different concepts accept the same arguments (besides the first one)? Supporting that solution seems like an unwise investment of time.

Implementation

A prototype was implemented last year in a branch of Clang. The branch is not actively maintained or available on compiler-explorer while we focus on both improving the implementation of Clang and upstreaming an implementation of P2841R7 [3]. Note that this implementation effort did not cover some aspects, particularly mangling.

Wording

🔲 Names of template specializations [temp.names]

template-argument:
constant-expression
type-id
nested-name-specifier$_{opt}$ *template-name*
nested-name-specifier `template` *template-name*
partially-applied-concept-argument

🔲 Template arguments [temp.arg]

🔲 Template template arguments [temp.arg.template]

[*Editor's note:* Add a subsection at the end of [temp.arg.template].]

🔲 Partially applied concept arguments [temp.arg.concept.partial]

partially-applied-concept-argument:
`concept` *nested-name-specifier*$_{opt}$ *concept-name* < *template-argument-list*$_{opt}$ >

The component names of a *partially-applied-concept-argument* are its *concept-name* and those of its *nested-name-specifier* (if any).

If *concept-name* denotes a template parameter pack, the program is ill-formed.

A *partially-applied-concept-argument* names an invented concept `X` defined as

```
template <Parameter Injected>
concept X = concept-name<Injected, template-argument-list>;
```

where `Parameter` is the first *template-parameter* in the *template-head* `H` of the concept designated by *concept-name*, without the ellipsis (if any).

If `H` declares a single non-pack template parameter or if the *constraint-expression* of `X` is not valid, the program is ill-formed.

[*Example:*

```
template <typename T, template <typename> concept... Concepts>
concept all_of  = (Concepts<T> && ...);

template <typename, auto>
concept A = true;

template <typename T, typename>
concept C = true;

template <typename... T>
concept D = true;

template <typename>
concept E = true;

void f(all_of<concept C<0>, concept C<int>, concept D<int>> auto); // ok
void f(all_of<concept E<int>> auto); // error: E declares a single non-pack template parameter

void f(all_of<concept C<int, int>> auto); // error: the constraint-expression of the invented
     concept would be C<T, int, int>, which is not a valid expression.
```

— end example]

? Type equivalence [temp.type]

Two *template-id*s are the same if

- their *template-name*s, *operator-function-id*s, or *literal-operator-id*s refer to the same template, and
- their corresponding type *template-argument*s are the same type, and
- the template parameter values determined by their corresponding constant template arguments [temp.arg.nontype] are template-argument-equivalent (see below), and
- their corresponding template *template-argument*s refer to the same template or are partial-concept-argument-equivalent.

[*Editor's note:* Add a subsection at the end of [temp.type].]

Two template *template-argument*s are *partial-concept-argument-equivalent* if

- they are both *partially-applied-concept-argument*,
- they have equivalent *template-argument-list*, and

- their *concept-name* refer to the same concept.

Feature test macro

[*Editor's note:* In [tab:cpp.predefined.ft], bump the value of `__cpp_template_parameters` to the date of adoption.]

Acknowledgments

Thanks to Lori Hughes for reviewing this paper and providing editorial feedback.

References

[1] Corentin Jabot and Gašper Ažman. P2841R0: Concept template parameters. https://wg21.link/p2841r0, 5 2023.

[2] Corentin Jabot and Gašper Ažman. P2989R2: A simple approach to universal template parameters. https://wg21.link/p2989r2, 6 2024.

[3] Corentin Jabot, Gašper Ažman, James Touton, and Hubert Tong. P2841R7: Concept and variable-template template-parameters. https://wg21.link/p2841r7, 2 2025.

[N5008] Thomas Köppe *Working Draft, Standard for Programming Language C++* https://wg21.link/N5008