

Pack Indexing for Template Names

Document #:	P3670R3
Date:	2026-06-11
Programming Language C++	
Audience:	CWG
Reply-to:	Corentin Jabot < corentin.jabot@gmail.com >

Revisions

R3

- Wording fixes following CWG review.

R2

- Apply the changes made by [CWG3027](#) [1] to the wording section.
- Wording fixes following CWG review.

R1

- Fix example.

Motivation

We added the ability to index packs of types and expressions in C++26 through [P2662R3](#) [4]. ([P2662R3](#) [4] is now implemented in Clang and GCC, and we got very positive feedback).

However, [P2662R3](#) [4] does not allow the indexing of a pack of templates. There is no good reason for that. The intent was always to be able to index all packs.

Both [P2841R7](#) [3] and [P2989R2](#) [2] were in flight, and it was not clear to me if either these papers would impact the indexing of packs of *template-names*. So, I punt that question to the present paper. It turns out that [P2841R7](#) [3] has no impact on the design of this paper - except that indexing a pack of concept template parameter just works - and [P2989R2](#) [2] was not approved for C++26.

In short, we are proposing to complete the design of pack indexing.

Design

The syntax for indexing a pack of template-name is similar to the syntax to the syntax used to index a pack of types or expressions.

```
template < template <typename> typename... TT>
struct S {
    template <typename T>
    using First = TT...[0]<T>;
};
```

The indexed pack is a *template-name* and can be used anywhere any *template-name* would be usable. All packs of template template parameters can be indexed (type, variable, concepts).

Implementation

This paper has not been implemented, but I am confident this can be implemented in Clang without trouble. I can't comment on other implementations.

Wording

◆ Argument-dependent name lookup [basic.lookup.argdep]

[*Note:* For purposes of determining (during parsing) whether an expression is a *postfix-expression* for a function call, the usual name lookup rules apply. In some cases a name followed by < is treated as a *template-name* even though name lookup did not find ~~a *template-name*~~ [the declaration of a template](#) (see [temp.names]).

[...]

— *end note*]

◆ Names of template specializations [temp.names]

A template specialization[temp.spec] can be referred to by a *template-id*:

simple-template-id:

template-name < *template-argument-list*_{opt} >

template-id:

simple-template-id

operator-function-id < *template-argument-list*_{opt} >

literal-operator-id < *template-argument-list*_{opt} >

template-name:

~~identifier~~

[simple-template-name](#)

[pack-index-template-name](#)

[pack-index-template-name](#):

simple-template-name ... [*constant-expression*]

[simple-template-name](#):

identifier

template-argument-list:
template-argument ... *opt*
template-argument-list , *template-argument* ... *opt*

template-argument:
constant-expression
type-id
nested-name-specifier *opt* *template-name*
nested-name-specifier *template* *template-name*

The component name of a *simple-template-id*, *template-id*, or *template-name* is the first name in it.

The *simple-template-name* *P* in a *pack-index-template-name* shall denote a pack.

The *constant-expression* shall be a converted constant expression [expr.const.const] of type `std::size_t` whose value *V*, termed the index, is such that $0 \leq V < \text{sizeof} \dots (P)$.

A *pack-index-template-name* is a pack expansion [temp.variadic].

[Note: The *pack-index-template-name* denotes the *V*th *template-name* of the pack. — end note]

A < is interpreted as the delimiter of a *template-argument-list* if **either**

- it follows a *splice-specifier* that either
 - appears in a type-only context or
 - is preceded by *template* or *typename*, or
- it follows a *pack-index-template-name*, or
- it follows a name that is not a *conversion-function-id* and
 - that follows the keyword *template* or a ~ after a *nested-name-specifier* or in a class member access expression, or
 - for which name lookup finds the injected-class-name of a class template or finds any declaration of a template, or
 - that is an unqualified name for which name lookup either finds one or more functions or finds nothing, or
 - that is a terminal name in a *using-declarator* [namespace.udcl], in a *declarator-id* [dcl.meaning], or in a type-only context other than a *nested-name-specifier* [temp.res].



Alias templates

[temp.alias]

A *template-declaration* in which the *declaration* is an *alias-declaration* [dcl.pre] declares the *identifier* to be an *alias template*. An alias template is a name for a family of types. The name of the alias template is a *simple-template-name*.

❖ Variadic templates

[temp.variadic]

[...]

A *pack expansion* consists of a *pattern* and an ellipsis, the instantiation of which produces zero or more instantiations of the pattern in a list (described below). The form of the pattern depends on the context in which the expansion occurs. Pack expansions can occur in the following contexts:

[...]

- In a *pack-index-expression*; the pattern is an *identifier*.
- In a *pack-index-specifier*; the pattern is a *typedef-name*.
- In a *pack-index-template-name*; the pattern is a *simple-template-name*.
- In a *fold-expression* [expr.prim.fold]; the pattern is the *cast-expression* that contains an unexpanded pack.
- In a fold expanded constraint[temp.constr.fold]; the pattern is the constraint of that fold expanded constraint.

[...]

The instantiation of a pack expansion considers items $E_1, E_2, \dots, E_N$, where N is the number of elements in the pack expansion parameters. Each E_i is generated by instantiating the pattern and replacing each pack expansion parameter with its i^{th} element. Such an element, in the context of the instantiation, is interpreted as follows:

[...]

When N is zero, the instantiation of a pack expansion does not alter the syntactic interpretation of the enclosing construct, even in cases where omitting the pack expansion entirely would otherwise be ill-formed or would result in an ambiguity in the grammar.

The instantiation of a `sizeof... expression`[expr.sizeof] produces an integral constant with value N .

When instantiating a *pack-index-expression* P , let K be the index of P . The instantiation of P is the *id-expression* E_K .

When instantiating a *pack-index-specifier* P , let K be the index of P . The instantiation of P is the *typedef-name* E_K .

When instantiating a *pack-index-template-name* P , let K be the index of P . The instantiation of P is the *simple-template-name* E_K .

❖ Type equivalence

[temp.type]

Two *template-ids* are the same if

- their *template-names*, *operator-function-ids*, or *literal-operator-ids* refer to the same template, and
- their corresponding type *template-arguments* 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-arguments* refer to the same template.

Two *template-ids* that are the same refer to the same class, function, or variable.

[...]

If an expression *e* is type-dependent [temp.dep.expr], `decltype(e)` denotes a unique dependent type. Two such *decltype-specifiers* refer to the same type only if their *expressions* are equivalent[temp.over.link]. [Note: However, such a type might be aliased, e.g., by a *typedef-name*. — end note]

[Editor's note: Both the black and green text below presumes that CWG3027 [1] has been approved and applied]

For a type template parameter pack *T*, *T* . . . [*constant-expression*] denotes a unique dependent type.

Two such *pack-index-specifiers* refer to the same type only if

- their *typedef-names* refer to the same template parameter pack and
- if neither of their *constant-expressions* is value-dependent, then they have the same value, otherwise their *constant-expressions* are equivalent ([temp.over.link]).

Two *pack-index-template-names* refer to the same template only if

- their *simple-template-names* refer to the same template parameter pack and
- if neither of their *constant-expressions* is value-dependent, then they have the same value, otherwise their *constant-expressions* are equivalent ([temp.over.link]).



Keywords

[gram.key]

New context-dependent keywords are introduced into a program by `typedef`[dcl.typedef], `namespace`[namespace.def], `class`[class], `enumeration`[dcl.enum], and `template`[temp] declarations.

```
typedef-name:
    identifier
    simple-template-id

namespace-name:
    identifier
    namespace-alias

namespace-alias:
    identifier
```

class-name:
 identifier
 simple-template-id
enum-name:
 identifier
simple-template-name:
 identifier

Feature test macros

[*Editor's note: Bump `__cpp_pack_indexing` to the date of adoption*] .

Bibliography

- [1] Corentin Jabot. CWG3027: Equivalence of pack-index-specifiers. <https://wg21.link/cwg3027>, 4 2025.
- [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.
- [4] Corentin Jabot and Pablo Halpern. P2662R3: Pack indexing. <https://wg21.link/p2662r3>, 12 2023.
- [N5008] Thomas Köppe *Working Draft, Standard for Programming Language C++* <https://wg21.link/N5008>