Pack Indexing for Template Names

Document #:	P3670R4
Date:	2026-06-12
Programming Language C++	
Audience:	CWG

Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Revisions

R4

- Wording fixes following CWG review (Adjusted [temp.deduct.guide] and [dcl.type.simple]

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-name*s. 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

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

[*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*]

[dcl.type.simple] Simple type specifiers

The simple type specifiers are

simple-type-specifier:
nested-name-specifier$_{opt}$ *type-name*
nested-name-specifier `template` *simple-template-id*
computed-type-specifier
placeholder-type-specifier
nested-name-specifier$_{opt}$ *simple-template-name*
pack-index-template-name
`char`
`char8_t`
`char16_t`
`char32_t`
`wchar_t`
`bool`
`short`
`int`
`long`
`signed`
`unsigned`
`float`
`double`
`void`

The component names of a *simple-type-specifier* are those of its *nested-name-specifier*, *type-name*, *simple-template-id*, *simple-template-name* and/or *type-constraint* (if it is a *placeholder-type-specifier*). The component name of a *type-name* is the first name in it.

A *placeholder-type-specifier* is a placeholder for a type to be deduced[dcl.spec.auto]. A *type-specifier* is a placeholder for a deduced class type[dcl.type.class.deduct] if ~~either~~

- it is of the form `typename`$_{opt}$ *nested-name-specifier*$_{opt}$ *simple-template-name* , or
- it is of the form `typename`$_{opt}$ *pack-index-template-name*, or
- it is of the form `typename`$_{opt}$ *splice-specifier* and the *splice-specifier* designates a class template or alias template.

The *nested-name-specifier* or *splice-specifier*, if any, shall be non-dependent and the *simple-template-name*, *pack-index-template-name* , or *splice-specifier* shall designate a deducible template. A *deducible template* is

- a class template,
- a type template template parameter, or
- an alias template `A` whose *defining-type-id* is of the form

  typename$_{opt}$ *nested-name-specifier*$_{opt}$ `template`$_{opt}$ *simple-template-id*

  where the *nested-name-specifier* (if any) is non-dependent and the *template-name* of the *simple-template-id* names a deducible template other than a type template template parameter of `A`.

[*Note:* An injected-class-name is never interpreted as a *template-name* in contexts where class template argument deduction would be performed[temp.local]. — *end note*] The other *simple-*

*type-specifier*s specify either a previously-declared type, a type determined from an expression, or one of the fundamental types[basic.fundamental]. [dcl.type.simple] summarizes the valid combinations of *simple-type-specifier*s and the types they specify.

🔲 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 $0 \leq V < \texttt{sizeof...}(P)$.

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.udecl], 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 $\mathtt{E}_1, \mathtt{E}_2, \ldots, \mathtt{E}_N$, where N is the number of elements in the pack expansion parameters. Each $\mathtt{E}_i$ 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* $\texttt{E}_K$.

When instantiating a *pack-index-specifier* P, let K be the index of P. The instantiation of P is the *typedef-name* $\texttt{E}_K$.

When instantiating a *pack-index-template-name* P, let K be the index of P. The instantiation of P is the *simple-template-name* $\texttt{E}_K$.

[?] 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.

Two *template-id*s 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-specifier*s refer to the same type only if their *expression*s 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-specifier*s refer to the same type only if

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

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

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

Deduction guides [temp.deduct.guide]

Deduction guides are used when a *template-name* or *splice-type-specifier* appears as a type specifier for a deduced class type[dcl.type.class.deduct]. Deduction guides are not found by name lookup. Instead, when performing class template argument deduction[over.match.class.deduct], all reachable deduction guides declared for the class template are considered.

deduction-guide:
explicit-specifier$_{opt}$ *simple-template-name* `(` *parameter-declaration-clause* `)` `->` *simple-template-id requires-clause*$_{opt}$ `;`

[*Example:*

```
template<class T, class D = int>
struct S {
        T data;
};
template<class U>
S(U) -> S<typename U::type>;

struct A {
        using type = short;
        operator type();
};
S x{A()};           // x is of type S<short, int>
```

— end example]

The same restrictions apply to the *parameter-declaration-clause* of a deduction guide as in a function declaration[dcl.fct], except that a generic parameter type placeholder[dcl.spec.auto] shall not appear in the *parameter-declaration-clause* of a deduction guide. The *simple-template-id* shall name a class template specialization. The *simple-template-name* shall be the same *identifier* as the *template-name* of the *simple-template-id*. A *deduction-guide* shall inhabit the scope to which the corresponding class template belongs and, for a member class template, have the same access. Two deduction guide declarations for the same class template shall not have equivalent *parameter-declaration-clause*s if either is reachable from the other.

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