Pack Indexing for Template Names

Document #:	P3670R0
Date:	2025-04-16
Programming Language C++	
Audience:	EWG

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

Motivation

We added the ability to index packs of types and expressions in C++26 through P2662R3 [3]. (P2662R3 [3] is now implemented in Clang and GCC, and we got very positive feedback).

However, P2662R3 [3] 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 [2] and P2989R2 [1] 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 [2] has no impact on the design of this paper - except that indexing a pack of concept template parameter just works - and P2989R2 [1] 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 {
    using First = TT...[0];
};
```

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

🔲 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] of type `std::size_t` whose value V, termed the index, is such that $0 \leq V <$ `sizeof...(`P`)`.

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

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

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

🔲 Variadic templates [temp.variadic]

In a template parameter pack that is a pack expansion [temp.param]:

- In a `sizeof...` expression[expr.sizeof]; the pattern is an *identifier*.
- 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.

[*Editor's note:* [...]]

The instantiation of a pack expansion considers items $\texttt{E}_1, \texttt{E}_2, \ldots, \texttt{E}_N$, where N is the number of elements in the pack expansion parameters. Each $\texttt{E}_i$ 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:

- if the pack is a template parameter pack, the element is
  - a *typedef-name* for a type template parameter pack,
  - an *id-expression* for a constant template parameter pack, or
  - a *template-name* for a template template parameter pack

  designating the i^{th} corresponding type, constant, or template template argument;
- if the pack is a function parameter pack, the element is an *id-expression* designating the i^{th} function parameter that resulted from instantiation of the function parameter pack declaration;
- if the pack is an *init-capture* pack, the element is an *id-expression* designating the variable introduced by the i^{th} *init-capture* that resulted from instantiation of the *init-capture* pack declaration; otherwise
- if the pack is a structured binding pack, the element is an *id-expression* designating the i^{th} structured binding in the pack that resulted from the structured binding declaration.

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* $\texttt{E}_K$.

[*Editor's note:* [...]]

🔲 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.

[*Editor's note:* [...]]

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*]

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

If the *constant-expression* of a *pack-index-specifier* is value-dependent, two such *pack-index-specifier* s refer to the same type only if their *constant-expression* s are equivalent [temp.over.link]. Otherwise, two such *pack-index-specifier* s refer to the same type only if their indexes have the same value.

If the *constant-expression* of a *pack-index-template-name* is value-dependent, two such *pack-index-template-name*s refer to the same template only if their *constant-expression*s are equiv-

alent [temp.over.link]. Otherwise, two such *pack-index-template-name*s refer to the same template only if their indexes have the same value.

🔲 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

template-name:
~~*identifier*~~
simple-template-name
pack-index-template-name

Feature test macros

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

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

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

[3] 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