Pack Indexing

Document #:	P2662R0
Date:	2022-10-15
Programming Language C++	
Audience:	EWG

Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>
	Pablo Halpern <phalpern@halpernwightsoftware.com>
	John Lakos <jlakos@bloomberg.net>
	Alisdair Meredith <ameredith1@bloomberg.net>
	Joshua Berne <jberne4@bloomberg.net>

Abstract

This paper expands on the pack indexing feature described in P1858R2 [4] and provides wording.

Revisions

R0

Initial revision

Motivation

The motivation for pack indexing is covered in ”Generalized pack declaration and usage” (P1858R2 [4]) and ”A plan for better template meta programming facilities in C++26” [P2632R0].

The short version is that packs are sequences of types/expressions, and indexing is a fundamental operation on sequences. C++ and its users have so far relied on deduction or library facilities such as `index_sequence` or full-fledged template metaprogramming libraries such as mp11 and boost.Hana to extract the Nth element of a pack, which has a high cost both in terms of code complexity and compiler throughput.

Previous works in this area also include P0565R0 [1], P1803R0 [2], N3761 [3] and N4235 [5].

Proposal

This paper proposes a new code language syntax to index packs of types (yielding a type), and packs of expressions (yielding an expression).

Syntax

The general syntax is *name-of-a-pack* `...` `[constant-expression]`. The syntax has the benefit of reusing familiar elements (`...` usually denotes a pack expansion) and `[]` subscripts, It is therefore natural that indexing a pack expansion reuses these elements.

```
template <typename... T>
constexpr auto first_plus_last(T... values) ->  T...[0] {
    return T...[0](values...[0] + values...[sizeof...(values)-1]);
}
int main() {
    //first_plus_last(); // ill-formed
    static_assert(first_plus_last(1, 2, 10) == 11);
}
```

Pack Index

The index of a pack indexing expression or specifier is an integral constant expression between 0 and `sizeof...(pack)`. Empty packs can't be indexed.

This paper does not add support for indexing from the end (as an alias of `T...[sizeof...(T)-N]`). Indeed, a negative index (`T...[-1]`) would be surprising.

Consider:

```
// Return the index of the first type convertible to Needle in Pack
or -1 if Pack does not contain a suitable type.
template <typename Needle, typename... Pack>
auto find_convertible_in_pack;

 // if find_convertible_in_pack<Foo, Types> is -1, T will be the last type, erronously.
using T = Types...[find_convertible_in_pack<Foo, Types>];
```

In general, incorrect computations in an index can lead to a negative value that should make the program ill-formed but would instead yield an incorrect type.

Note however that circle does support from-the end indexing using a negative index, and Sean Baxter reports no surprise from using this feature.

The solution for indexing from the end is to provide a specific syntax, for example, C# uses `^` to mean "from the end", and Dlang interprets `$` as the size of the array it appears in

```
using Foo = T...[0];
using Bar = T...[^1]; // First from the end
using Bar = T...[$ - 1]; // First from the end
```

Given that there are different alternatives, all of which can be added later, and for which we do not have usage experience, this proposal does not support from-the-end indexing.

Indexing a pack of types

Indexing a pack of type is a type specifier that can, like `decltype` appear:

- As a simple-type-specifier
- As a base class specifier
- As a nested name specifier
- As the type of an explicit destructor call

Type deduction

Pack indexing specifiers should not allow deducing the pack from such an expression.

Consider:

```
template <typename... T>
void f(T...[0]);
f(0);
```

It doesn't really make sense to start thinking about how deduction would work here or what that code would possibly mean. We simply always consider packs indexing non-deduced contexts.

Indexing a pack of expressions

The intent is that a pack indexing expression behaves exactly as the underlying expression would. In particular, `decltype(id-dexpression)` and `decltype(pack-indexing-expression)` behave the same.

Future Evolutions

The syntax can be extended in subsequent proposals to support:

- Indexing packs introduced by structured bindings or other non dependent packs
- Indexing packs of template template parameters
- From-the-end-indexing
- Pack Slicing (Returning a subset of a pack as an unexpanded packs)
- Packs of univeral template parameters could be indexed in the same way.

Potential impact on existing code

In C++11, `T... [N]` declares a pack of arrays of size `N`;

```
template <typename... T>
void f(T... [1]); //
int main() {
    f<int, double>(nullptr, nullptr); // void f<int, double>(int [1], double [1])
}
```

Neither MSVC nor gcc supports this syntax and this pattern does not appear outside of compiler test suites (from a search on Github, isocpp and in VCPKG).

Should anyone be affected, a workaround is to name the variable.

```
template <typename... T>
void f(T... foo[1]);
```

Implementation

This proposal is inspired from features implemented in the Circle compiler (with the same syntax). The provided wording is based on an implementation in a fork of clang, which is available on compiler explorer.

The implementation is available on Compiler Explorer.

As of the writting of the paper, the implementation is known not to support pack indexing inside of the expansion of another template parameters. We hope to fix these issues before Kona.

Ie, the following is intended to work but currently does not

```
template <typename... Ts, int... N>
auto f(Ts... ts) {
    return std::tuple{ (ts...[N])... };
}
```

Wording

🞐 Qualified name lookup [basic.lookup.qual]

🞐 General [basic.lookup.qual.general]

Lookup of an *identifier* followed by a `::` scope resolution operator considers only namespaces, types, and templates whose specializations are types. If a name, *template-id*, pack-indexing-specifier, or *decltype-specifier* is followed by a `::`, it shall designate a namespace, class, enumeration, or dependent type, and the `::` is never interpreted as a complete *nested-name-specifier*.

Names [expr.prim.id]

Unqualified names [expr.prim.id.unqual]

unqualified-id:
identifier
pack-indexing-expression
operator-function-id
conversion-function-id
literal-operator-id
~ *type-name*
~ *decltype-specifier*
~ pack-indexing-specifier
template-id

An *identifier* is only an *id-expression* if it has been suitably declared or if it appears as part of a *declarator-id*. An *identifier* that names a coroutine parameter refers to the copy of the parameter. [*Note:* For *operator-function-id*s, see **??**; for *conversion-function-id*s, see **??**; for *literal-operator-id*s, see **??**; for *template-id*s, see **??**. A *type-name* *, pack-index-type,* or *decltype-specifier* prefixed by ~ denotes the destructor of the type so named; see **??**. Within the definition of a non-static member function, an *identifier* that names a non-static member is transformed to a class member access expression. — *end note*]

A *component name* of an *unqualified-id* U is

- U if it is a name or
- the component name of the *template-id* or *type-name* of U, if any.

[*Note:* Other constructs that contain names to look up can have several component names. — *end note*] The *terminal name* of a construct is the component name of that construct that appears lexically last.

Qualified names [expr.prim.id.qual]

qualified-id:
nested-name-specifier `template`$_{opt}$ *unqualified-id*

nested-name-specifier:
::
type-name ::
namespace-name ::
decltype-specifier ::
pack-indexing-specifier ::
nested-name-specifier identifier ::
nested-name-specifier `template`$_{opt}$ *simple-template-id* ::

The component names of a *qualified-id* are those of its *nested-name-specifier* and *unqualified-id*. The component names of a *nested-name-specifier* are its *identifier* (if any) and those of its *type-name*, *namespace-name*, *simple-template-id*, and/or *nested-name-specifier*.

A *nested-name-specifier* is *declarative* if it is part of

- a *class-head-name*,
- an *enum-head-name*,
- a *qualified-id* that is the *id-expression* of a *declarator-id*, or
- a declarative *nested-name-specifier*.

A declarative *nested-name-specifier* shall not have a *decltype-specifier*. A declaration that uses a declarative *nested-name-specifier* shall be a friend declaration or inhabit a scope that contains the entity being redeclared or specialized.

The *nested-name-specifier* `::` nominates the global namespace. A *nested-name-specifier* with a *decltype-specifier* nominates the type denoted by the *decltype-specifier*, which shall be a class or enumeration type. If a *nested-name-specifier* N is declarative and has a *simple-template-id* with a template argument list A that involves a template parameter, let T be the template nominated by N without A. T shall be a class template.

- If A is the template argument list of the corresponding *template-head* H, N nominates the primary template of T; H shall be equivalent to the *template-head* of T.
- Otherwise, N nominates the partial specialization of T whose template argument list is equivalent to A; the program is ill-formed if no such partial specialization exists.

Any other *nested-name-specifier* nominates the entity denoted by its *type-name*, *namespace-name*, *identifier*, or *simple-template-id*. If the *nested-name-specifier* is not declarative, the entity shall not be a template.

A *qualified-id* shall not be of the forms *nested-name-specifier* `template`$_{opt}$ `~` *decltype-specifier* ~~nor of the form~~ , *decltype-specifier* `::` `~` *type-name* , *nested-name-specifier* `template`$_{opt}$ `~` *pack-indexing-specifier* or *pack-indexing-specifier* `::` `~` *type-name* .

The result of a *qualified-id* Q is the entity it denotes. The type of the expression is the type of the result. The result is an lvalue if the member is

- a function other than a non-static member function,
- a non-static member function if Q is the operand of a unary `&` operator,
- a variable,
- a structured binding, or
- a data member,

and a prvalue otherwise.

[Editor's note: Add a new section after [expr.prim.id.qual]]

🔲 Pack Indexing Expression [expr.prim.pack.index]

pack-indexing-expression:
id-expression `...` `[` *constant-expression* `]`

The *id-expression* in a *pack-indexing-expression* shall denote a pack.

The *constant-expression* shall be an integral constant expression. The *constant-expression* shall evaluate to a value V such that 0 <= V < `sizeof...(id-expression)`.

The index of a *pack-indexing-expression* is the value of its *constant-expression*.

A *pack-indexing-expression* denotes the $index^{\text{th}}$ expression in a pack ([temp.variadic]). A *pack-indexing-expression* is a pack expansion ([temp.variadic]).

🔲 Simple type specifiers [dcl.type.simple]

The simple type specifiers are

simple-type-specifier:
nested-name-specifier$_{opt}$ *type-name*
nested-name-specifier `template` *simple-template-id*
decltype-specifier
pack-indexing-specifier
placeholder-type-specifier
nested-name-specifier$_{opt}$ *template-name*

[...]

Table 1: *simple-type-specifier*s and the types they specify

Specifier(s)	Type
type-name	the type named
simple-template-id	the type as defined in [temp.names]
pack-indexing-specifier	the type as defined in [dcl.type.pack.indexing]
decltype-specifier	the type as defined in [dcl.type.decltype]
placeholder-type-specifier	the type as defined in [dcl.spec.auto]
template-name	the type as defined in [dcl.type.class.deduct]
`char`	"`char`"
`unsigned char`	"`unsigned char`"
`signed char`	"`signed char`"
`char8_t`	"`char8_t`"
`char16_t`	"`char16_t`"

When multiple *simple-type-specifier*s are allowed, they can be freely intermixed with other *decl-specifier*s in any order. [*Note:* It is implementation-defined whether objects of `char` type are represented as signed or unsigned quantities. The `signed` specifier forces `char` objects to be signed; it is redundant in other contexts. — *end note*]

[Editor's note: Add a new section after [dcl.type.simple]]

🚧 Pack Indexing Specifier [dcl.type.pack.indexing]

pack-indexing-specifier:
typedef-name `...` `[` *constant-expression* `]`

The *typedef-name* in a *pack-indexing-specifier* shall denote a pack.

The *constant-expression* shall be an integral constant expression. The *constant-expression* shall evaluate to a value V such that 0 <= V < `sizeof...(typedef-name)`.

The index of a *pack-indexing-specifier* is the value of its *constant-expression*.

A *pack-indexing-specifier* denotes the type of the $index^{\text{th}}$ `typedef-name` in a pack ([temp.variadic]). A *pack-indexing-specifier* is a pack expansion ([temp.variadic]).

🚧 Decltype specifiers [dcl.type.decltype]

decltype-specifier:
`decltype` `(` *expression* `)`

For an expression E, the type denoted by `decltype(`E`)` is defined as follows:

- if E is an unparenthesized *id-expression* naming a structured binding, `decltype(`E`)` is the referenced type as given in the specification of the structured binding declaration;
- otherwise, if E is an unparenthesized *id-expression* naming a non-type *template-parameter*, `decltype(`E`)` is the type of the *template-parameter* after performing any necessary type deduction;
- otherwise, if E is an unparenthesized *id-expression* or an unparenthesized class member access, `decltype(`E`)` is the type of the entity named by E. If there is no such entity, the program is ill-formed;

  [*Note:* A *pack-indexing-expression* is an *id-expression*. — *end note*]
- otherwise, if E is an xvalue, `decltype(`E`)` is `T&&`, where `T` is the type of E;
- otherwise, if E is an lvalue, `decltype(`E`)` is `T&`, where `T` is the type of E;
- otherwise, `decltype(`E`)` is the type of E.

The operand of the `decltype` specifier is an unevaluated operand.

[*Example:*

```
const int&& foo();
int i;
struct A { double x; };
const A* a = new A();
decltype(foo()) x1 = 17;        // type is const int&&
decltype(i) x2;                 // type is int
decltype(a->x) x3;              // type is double
decltype((a->x)) x4 = x3;       // type is const double&
```

```
[](auto... pack){
    decltype(pack...[0]); // // type is int
    decltype((pack...[0]));  // type is int&
}(0);
```

— *end example*]

🔲 Classes [class]

🔲 Destructors [class.dtor]

In an explicit destructor call, the destructor is specified by a ~ followed by a *type-name* , *pack-index-specifier*, or *decltype-specifier* that denotes the destructor's class type. The invocation of a destructor is subject to the usual rules for member functions; that is, if the object is not of the destructor's class type and not of a class derived from the destructor's class type (including when the destructor is invoked via a null pointer value), the program has undefined behavior.

🔲 Derived classes [class.derived]

🔲 General [class.derived.general]

A list of base classes can be specified in a class definition using the notation:

base-clause:
: *base-specifier-list*

base-specifier-list:
base-specifier ...$_{opt}$
base-specifier-list , *base-specifier* ...$_{opt}$

base-specifier:
attribute-specifier-seq$_{opt}$ *class-or-decltype*
attribute-specifier-seq$_{opt}$ `virtual` *access-specifier*$_{opt}$ *class-or-decltype*
attribute-specifier-seq$_{opt}$ *access-specifier* `virtual`$_{opt}$ *class-or-decltype*

class-or-decltype:
nested-name-specifier$_{opt}$ *type-name*
nested-name-specifier `template` *simple-template-id*
decltype-specifier
pack-index-specifier

access-specifier:
`private`
`protected`
`public`

Type equivalence [temp.type]

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

For a template argument `T`, if the *constant-expression* of a *pack-index-specifier* is type-dependent `T...[`*constant-expression*`]` denotes a unique dependent type. Two such *pack-index-specifier*s refer to the same type only if their *constant-expression*s are equivalent and their type are the same.

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 function parameter pack; the pattern is the *parameter-declaration* without the ellipsis.
- In a *using-declaration*; the pattern is a *using-declarator*.
- In a template parameter pack that is a pack expansion:
  - if the template parameter pack is a *parameter-declaration*; the pattern is the *parameter-declaration* without the ellipsis;
  - if the template parameter pack is a *type-parameter*; the pattern is the corresponding *type-parameter* without the ellipsis.
- In an *initializer-list*; the pattern is an *initializer-clause*.
- In a *base-specifier-list*; the pattern is a *base-specifier*.
- In a *mem-initializer-list* for a *mem-initializer* whose *mem-initializer-id* denotes a base class; the pattern is the *mem-initializer*.
- In a *template-argument-list*; the pattern is a *template-argument*.
- In an *attribute-list*; the pattern is an *attribute*.
- In an *alignment-specifier*; the pattern is the *alignment-specifier* without the ellipsis.
- In a *capture-list*; the pattern is the *capture* without the ellipsis.
- In a `sizeof...` expression; the pattern is an *identifier*.
- In a *pack-indexing-expression* ; the pattern is an *id-expression*.
- In a *pack-indexing-specifier* ; the pattern is an *identifier*.
- In a *fold-expression*; the pattern is the *cast-expression* that contains an unexpanded pack.

[*Example:*

```
template<class ... Types> void f(Types ... rest);
template<class ... Types> void g(Types ... rest) {
    f(&rest ...);     // ``&rest ...'' is a pack expansion; ``&rest'' is its pattern
}
```

— end example]

For the purpose of determining whether a pack satisfies a rule regarding entities other than packs, the pack is considered to be the entity that would result from an instantiation of the pattern in which it appears.

A pack whose name appears within the pattern of a pack expansion is expanded by that pack expansion. An appearance of the name of a pack is only expanded by the innermost enclosing pack expansion. The pattern of a pack expansion shall name one or more packs that are not expanded by a nested pack expansion; such packs are called *unexpanded packs* in the pattern. All of the packs expanded by a pack expansion shall have the same number of arguments specified. An appearance of a name of a pack that is not expanded is ill-formed. [*Example:*

```
template<typename...> struct Tuple {};
template<typename T1, typename T2> struct Pair {};

template<class ... Args1> struct zip {
    template<class ... Args2> struct with {
        typedef Tuple<Pair<Args1, Args2> ... > type;
    };
};

typedef zip<short, int>::with<unsigned short, unsigned>::type T1;
// T1 is Tuple<Pair<short, unsigned short>, Pair<int, unsigned>>
typedef zip<short>::with<unsigned short, unsigned>::type T2;
// error: different number of arguments specified for Args1 and Args2

template<class ... Args>
void g(Args ... args) {                   // OK, Args is expanded by the function parameter pack args
    f(const_cast<const Args*>(&args)...);   // OK, ``Args'' and ``args'' are expanded
    f(5 ...);                               // error: pattern does not contain any packs
    f(args);                                // error: pack ``args'' is not expanded
    f(h(args ...) + args ...);              // OK, first ``args'' expanded within h,
    // second ``args'' expanded within f
}
```

— end example]

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:

- if the pack is a template parameter pack, the element is an *id-expression* (for a non-type template parameter pack), a *typedef-name* (for a type template parameter pack declared without `template`), or a *template-name* (for a type template parameter pack declared with `template`), designating the i^{th} corresponding type or value 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; otherwise
- 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.

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 produces an integral constant with value N.

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

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

[...]

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

The non-deduced contexts are:

- The *nested-name-specifier* of a type that was specified using a *qualified-id*.
- A *pack-index-specifier,*
- 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 the *parameter-declaration-list*.

C++ and ISO C++23 [diff.cpp23]

Declarations [diff.cpp23.dcl.dcl]

Change: [decl.array]

Previously, `T...[`n`]` would declarare a pack of function parameters of type "array of `T` of size n". `T...[`n`]` is now a *pack-indexing-specifier*.
Rationale: Improve the handling of packs.
Effect on original feature: Valid C++ 2023 code that declares pack of arrays parameter without specifying a `declarator-id` may become ill-formed.

```
template <typename... T>
void f(T... [1]);
template <typename... T>
void g(T... ptr[1]);

int main() {
    f<int, double>(nullptr, nullptr); // ill-formed, previously void f<int, double>(int [1], double
[1])
    g<int, double>(nullptr, nullptr); // ok
}
```

Feature test macros

[Editor's note: Add a new macro in `[tab:cpp.predefined.ft]` : `__cpp_pack_indexing` set to the date of adoption] .

Acknowledgments

We would like to thank Bloomberg for sponsoring this work.

Sean Baxter for his work on Circle and Barry Revzin, for his work on P1858R2 [4], both works being the foundation of the design presented here.

Also Thanks to Lewis Baker, for his valuable feedback on this paper.

References

[1] Bengt Gustafsson. P0565R0: Prefix for operator as a pack generator and postfix operator[] for pack indexing. https://wg21.link/p0565r0, 2 2017.

[2] JeanHeyd Meneide. P1803R0: packexpr(args, i) - compile-time friendly pack inspection. https://wg21.link/p1803r0, 8 2019.

[3] Sean Middleditch. N3761: Proposing type_at<>. https://wg21.link/n3761, 8 2013.

[4] Barry Revzin. P1858R2: Generalized pack declaration and usage. https://wg21.link/p1858r2, 3 2020.

[5] Daveed Vandevoorde. N4235: Selecting from parameter packs. https://wg21.link/n4235, 10 2014.

[P2632R0] Corentin Jabot, Pablo Halpern, John Lakos, Alisdair Meredith, Joshua Berne, and Gašper Ažman
A plan for better template meta programming facilities in C++26
https://wg21.link/P2632R0
October 2022

[N4885] Thomas Köppe *Working Draft, Standard for Programming Language C++*
https://wg21.link/N4885