Pack Indexing

Document #: P2662R3
Date: 2023-11-08
Programming Language C++
Audience: EWG
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>
Pablo Halpern <phalpern@halpernwightsoftware.com>

Abstract

This paper expands on the pack indexing feature described in P1858R2 [8] and provides wording.

Revisions

R3

- Wording improvements, as reviewed by CWG in Kona

R2

- Wording improvements, as reviewed by CWG in Varna
- Add a section about ways types could be deduced from an index-type-specifier.
- expand the ”future evolutions” section.

R1

- At EWG’s request, we explained in more detail the syntax choices and explored alternatives.
- Wording improvements

R0

- Initial revision

Motivation

The motivation for pack indexing is covered in P1858R2 [8] and P2632R0 [P2632R0].

The short background version is that packs are sequences of types or 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.

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

Previous works in this area also include P0565R0 [2], P1803R0 [5], N3761 [6] and N4235 [12].

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. That indexing a pack expansion reuses these elements is, therefore, natural:

```
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);
}
```

This syntax is used by Circle and was initially proposed by P1858R2 [8].

Other syntactic options considered

The `pack...[index]` syntax was selected for this proposal after considering a number of other options, some of which have been proposed by other committee members.

- `pack.[index]`; see N4235 [12]
- `pack<index>` or `pack...<index>`
- `std::nth_type<index, pack...>` or `std::nth_value<index>(pack...)`
- `packexpr(args, I)`; see P1803R0 [5]
- `[index]pack`; see P0535R0 [14]
- Pack objects; see P2671R0 [9]

So, which syntax is the best choice?

Any syntax would be better than the status quo. However, having considered the different options, the original choice, `Pack...[N]`, still seems to be the best option, as it is straightforward and consistent with existing pack features. What follows is a detailed analysis of the different options, but readers who find the proposed syntax acceptable might want to skip forward to the ”Pack Index” section.

Before arguing what the best syntax is, which is ultimately subjective, we need to understand the constraints.

- An indexed pack can produce a type, an expression, and maybe in the future a template template parameter, a universal template, and so on; thus we need a syntax that can work in all contexts.
- We want a syntax that can be expanded to support slicing in the future.
- Pack elements can be array-like or tuple like, so we need to be careful about ambiguities. In particular, directly applying a subscript to a pack (`P[i]`, where `P` is a pack) is nonviable. Indeed, indexing a pack of arrays (`ArrayPack...[index]`), indexing each array in a pack (`ArrayPack[index]...`), indexing a single array with a pack of indexes (`Array[IndexPack]...`), and indexing each array in a pack of arrays with an index from an (equal-length) pack of indexes (`ArrayPack[IndexPack]...`) are different operations, all of which are useful. [This example] shows how, using the proposed syntax, one can distinguish between indexing the arrays from a pack with indexes from a pack (`ArrayPack[IndexPack]...`) and indexing a pack of arrays with a pack of indexes (`ArrayPack...[IndexPack`

`pack.[index]`

Historically in C++ (and C++-like languages), a single dot denotes member access. Reusing that syntax for pack indexing would introduce a semantics inconsistency and, more importantly, could close the door to future evolutions.

P1858R0 [7] proposes `tuple-like.[N]` as syntactic sugar over `get<N>(tuple-like)` and `aggregate.[N]` returning the Nth data member of an aggregate.

Ideally, `tuple[N]` would simply work. A few proposals have tried to improve the user-friendliness of tuple indexing (see P2726R0 [4] and P0311R0 [13]).

We see no technical limitation to making `tuple-like[N]` work on types that do not otherwise define a `operator[]`. P1858R2 [8] prefers `.[]` to `[]` because the paper proposes to index not only tuple-like but also any other decomposable types, such as aggregates. An aggregate might have an `operator[]` already, so disambiguating is unnecessary. Note that array-like classes are tuple-like, but their `operator[]` has the same semantics as what tuple indexing would do.

Whether tuple-indexing should be written as `.[]` or `[]` depends on whether indexing an aggregate is a frequent enough use case to warrant a specific syntax, rather than indexing the pack formed by unpacking an aggregate (`aggregate[:]` — in the syntax of P1858R2 [8], `aggregate.[N]` is a shorthand for `aggregate[:]...[N]`).

If a shorthand syntax to index the fields of an aggregate doesn’t seem useful, then we can index tuple-like objects with `tuple[N]`, which would make `pack.[N]` up for grabs. But that does not mean we should. We see little motivation — other than availability — for using syntax that usually denotes member access for packs.

Angle brackets

We could use angle brackets instead of square brackets, the argument, we suppose, being argument is that <> is more template-like and that pack indexing is also template-like. However, most languages have existing practices in which `[]` is to be used for both indexing and slicing. Being consistent with existing practice doesn't hurt. Besides, pack indexing will often occur in angle-bracket-heavy code, so using brackets for indexing too would not look better.

`std::nth_type<index, pack...>` or `std::nth_value<index>(pack...)`

As mentioned in the motivation section, there already exist library-only approaches to indexing a pack in mp11, boost.hana, and other libraries. Indeed, most implementations of the C++ Standard Library contain a private metafunction or two for this purpose.

The implementations are not complicated, but they are hard to write correctly and, in non-optimized compiles, can result in the generation of a large number of symbols and small functions:

```
template <size_t Index, class P0, class... Pack>
struct __nth_type_imp
{
    using type = typename __nth_type_imp<Index - 1, Pack...>::type;
};

template <class P0, class... Pack>
struct __nth_type_imp<0, P0, Pack...>
{
    using type = P0;
};

template <size_t Index, class P0, class... Pack>
using nth_type = typename __nth_type_imp<Index, P0, Pack...>::type;

template <size_t Index, class T0, class... Types>
constexpr decltype(auto) nth_value(T0&& p0, Types&&... pack) noexcept
{
    if constexpr (0 == Index)
        return std::forward<T0>(p0);
    else
        return nth_value<Index-1>(std::forward<Types>(pack)...);
}
```

The obvious advantage of this approach is that it is implemented entirely in the library, with no language changes necessary. However, the disadvantages are significant:

- Not only is the syntax harder to use but there are different syntaxes for packs of types versus packs of values. A metafunction for indexing a pack of templates (not shown) would have a third name and require yet another syntax.

- The recursion level for each of these facilities is $O(index)$. The instantiations for each index value is not re-used for other index values, so `nth_type<5, pack...>` and `nth_type<6, pack...>` produce 11 instantiations total, even with the same `pack`. Retrieving every element of a pack of size N, requires `O(N`^{2}`)` template instantiations. If implemented entirely as a library, the drag on compile time can be quite large.

  A compiler can reduce the instantiation expense through the use of an intrinsic, and several compilers have implemented such intrinsics. However, there is no guarantee that every implementation will do so. Moreover, even if the library template invoked an intrinsic, one level of pack expansion is still needed for the indirection, making the best-case scenario $(O(index))$.

- The library solution does not have a future path for treating a subset of a pack as an unexpanded pack (slicing). Because packs are not first-class objects or types, it is doubtful that any metafunction could yield an unexpanded pack without language changes, thus eliminating the advantages of a library-only approach.

- The library solution would also not work for universal template parameters, as described in P1985R1 [1].

Magic function

P1803R0 [5] proposed `packexpr(pack, N)`, i.e., `reserved-identifier(pack, N)`. We would need to find an identifier that is meaningful for all types of packs (not just expressions) and is not widely used. The identifier would most certainly have to be a globally reserved keyword (not a contextual one), as pack indexing can appear anywhere either a type, or an expression can appear, which is everywhere.

Perhaps `packelement(pack, index)` would work, but we would arguably need another identifier for slicing.

`[N]pack...`

P0535R0 [14] explored putting the index before the pack, and this method would probably work, although some looking ahead might be necessary to distinguish that syntax from that of lambdas. We see no logic to this choice other than, again, availability.

Pack objects

P2671R0 [9] proposes a syntax (rather arbitrarily given it's currently unused), to create a *pack object* or, rather, to instruct the compiler to manipulate a pack without expanding it. The one motivating use case is for the expansion statement, where that syntax allows us to distinguish *looping* over a tuple versus looping over a pack in a nonambiguous manner. Other examples, including pack indexing and slicing, look similar but arguably worse using this *pack object* mechanism.

As noted, an expansion statement can be used with a pack using

```
    template for (auto elem : std::tuple(ts...)) { ... }
```

Revzin observes this use is "wasteful," which is true, but a big part of the problem is that a tuple is a much heavier type than it ought to be, and pack indexing is one of the tools we need to make tuples lighter - along with forwarding references deduction and member packs.

In this model, `pack!` is a pack object (the syntax seems to have been chosen rather arbitrarily on the fact it's currently not used), and then that object can be indexed using `pack![N]`.

The pack object can be modeled by taking a reflection of the pack and then indexing that object splices the Nth element, as explained in P2671R0 [9]

```
template <std::vector<std::meta::info> V>
struct PackObject {
    constexpr auto operator[](std::ptrdiff_t idx) const {
        return [: V[idx] :];
    }
};
```

This model does not explain how it would deal with packs of types, template parameters, universal templates, and anything that is not an expression as an `operator[]` has to return an expression. Note that it could arguably return a meta::info and then let the user do the splicing itself, at which point any proclaimed syntax advantage would be lost.

We probably should discuss the concept of pack-objects, because they raise an interesting question: Do we need a syntax to reflect on a pack, that would be a shorthand for `std::vector{^ts...}`? Maybe? Because I think that's the question that "pack-objects" fundamentally tries to answer.

And it is true that pack indexing is equivalent to

```
 [: std::vector{^pack...}[N] :]
```

And slicing can probably be emulated with

```
...[: std::vector{^pack...} | std::views::drop(N) | std::views::take(M) ] :]...
```

Note that P2671R0 [9] seems to propose a slicing operator (`pack[N:M]`) anyway, because the code above is not exactly terse.

Barry observes that slicing through a pack object creates more questions than it solves. If slicing a pack creates a pack - which seems fairly obvious, does slicing a pack object create a pack object or a pack?

Both answers seem equally justified, however, if slicing a pack object produces a pack object, now we need another syntax to turn the pack object back into a pack, and this is how P2671R0 [9] ends up suggesting `ts![1:]~...` or `ts![1..]~...`.

Which is a lot of new syntax constructs that do try to offer a consistent story. And only work for expressions!

In terms of compile time, an implementation could either do what the paper proposes, ie create a pack object, which includes a constexpr vector, evaluate that, and splice it to get the resulting expression which would be less than optimal (and yet faster than any existing solution!), or an implementation could be cleaver and treat `![N]` as a single ”pack indexing” construct, which would be efficient but would be, in effect, a pack indexing operation spelled `![]` instead of `...[]`.

I will not claim that one is prettier than the other, but one is certainly a natural extension of the current grammar.

Something else?

We could entertain all sorts of syntaxes that are not yet used: two dots, four dots, `!`, `@` $, and so on. None of them would be a logical extension of the existing grammar, and since we are extending an existing facility, we should aim for something more justified than simply ”it's not yet used by something else.”

Too many dots?

One of the arguments heard against the `T...[N]` syntax is the ”too many dots” argument. And it's true that code that performs a lot of pack manipulation has numerous dots. However, the code does make sense and is readable; e.g., see this linked implementation of tuple. A few advantages come along with `...` for pack expansions.

- Seeing at a glance which pattern is expanded and where is useful.
- The syntax of pack declarations and pack expansion has so far been rather consistent and follows a given pattern. Multiple paper authors have come up with the same syntax independently because it's an obvious extension of existing syntax.
- Using `...`*`postfix-syntax`* for this feature — and future pack-related features (and nothing else) — gives us a clear, reserved design space for packs.

Allowable values for the pack index

The index of a pack indexing expression or specifier is an integral constant expression between 0 and `sizeof...(pack) - 1`. Empty packs can't be indexed.

In other proposals, a negative index, `-N`, would be interpreted as indexing from the end of the pack — as an alias of `T...[sizeof...(T)-N]`. However, a negative index could occur by accident, yielding surprising results:

```
// 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, erroneously.
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 surprises from using this feature.

An alternative 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:

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

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

Indexing a pack of types

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

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

Type deduction

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

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

However, CWG in Varna realized a way where a pack indexing type specifier could be deduced, and is asking EWG to chime in as changing something to be deducible after the fact is always going to be a breaking change.

Consider the following example, graciously provided by Jens Maurer:

```
template<class ...T>
int f( T...[0], std::tuple<T...> );
int x = f(-1, std::tuple(5u, 1));
```

Here `T...` is deduced from `5u, 1` to be `unsigned, int`, the first parameter of `f` is unsigned and `-1` is converted to unsigned.

This is what we propose.

But, we could imagine to instead, in that case, consider `T...` to be a sparse, infinitely-sized pack of a sort - as we don't know its `size`, deduced `T...[0]` to be `int` (from `-1`), then deduce the remaining of the pack (here the 2nd element) from the remaining arguments, and then check we have deduced a type for each element of the pack.

However, allowing parts of packs to be deduced only makes sense if the size of the pack is independently deduced. Consider:

```
template <typename... T>
int f(T...[0], int) {
    static_assert(sizeof...(T) == ???);
}
int a = f(0, 1);
```

Is the intent that `T...` is `int` (size 1) or `int, int` (size 2)? Both seem reasonable options. We could add a rule that the size of a deduced pack is deduced to be the size of the smallest pack that can be deduced, but this would surely not make sense in all cases, and the implementation costs seem impossible to justify.

So the only context in which a pack indexing specifier could be deduced seems to be when the entire pack is otherwise deduced, or when the size of a pack is deduced independently, which only ever happens for empty (non-indexable packs), which is awfully limited, and yet very complex.

And this is before we consider whether this would impact partial ordering rules, which it certainly would, for example, what should this do?

```
template <typename... T>
int f(T...[0], tuple<T...>);  // #1

template <typename F, typename... T>
int f(F, tuple<T...>);  // #2

int a = f(0, tuple{1});   // #1 or #2
```

We are sure there may be cases where this feature could be useful, yet we failed to come up with genuine motivation for it. The reason for this discussion is that it would be a breaking change to adopt something like that in a subsequent standard.

Going back to the first example, if `T...[0]` is not a deduced context, it will correspond to unsigned, and if we allow it to be deduced it will be deduced to `int` in the first parameter and in the second, which would be ill-formed.

In this proposal, we simply always consider pack indexing to be a non-deduced context.

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-index-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. This should just work without further modification when both this paper and P1061R5 [10] are both adopted.
- Indexing packs of template template parameters. To avoid confusing merge conflicts, we aim to progress concept, variable template, and universal template parameters before proposing indexing of template parameters packs. But our hope is to complete the feature for C++26 so that indexing works on all kinds of packs. Packs of universal template parameters could and should be indexed in the same way.

Other features are more evolutionary but this paper left the design open to allow coherent design of:

- From-the-end-indexing
- Pack slicing (returning a subset of a pack as an unexpanded pack) as discussed in P1858R0 [7] and P2632R0 [P2632R0].
- Indexing packs of arbitrary expressions, as described below.

Extending pack indexing to arbitrary expressions

In this proposal, we limit pack-index expressions to index a pack of *id-expression*s, which then denotes a function parameter pack, NTTP pack or, the pack introduced by a structured binding.

The reasons for that is that it is more readable and easier on the compiler to first index a pack, and then construct an expression around that, rather than construct an arbitrarily complex pattern, expand it, and reject all but one of the expanded elements - especially as substitution failures could occur.

Nevertheless, in the presence of multiple packs, being able to index arbitrary complex might improve code readability slightly, which would be the one motivation for this feature.

```
void g(auto&&);
template <typename...T>
void f(T&&... t) {
    g(std::forward<T...[0]>(t...[0])); // current proposal
    g(std::forward<T>(t)...[0]); // not proposed nor implemented
}
```

This is a possible evolution of this feature that can be explored but we should understand what would be the viable implementation strategies so that a compiler can avoid instantiating expressions that are not used.

Potential impact on existing code

In C++23, `T... [N]` is a valid syntax for declaring a function parameter matching a pack of unnamed 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). The fact that 2 major compilers did not implement this syntax in over a decade is indicative of its lack of usefulness.

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

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

See this linked demonstration.

Implementation

This proposal is inspired by 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.

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*, or ~~*decltype-specifier*~~ *computed-type-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]

� General [expr.prim.id.general]

id-expression:
    unqualified-id
    qualified-id
    pack-index-expression

� Unqualified names [expr.prim.id.unqual]

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

An *identifier* is only an *id-expression* if it has been suitably declared [dcl.dcl] or if it appears as part of a *declarator-id* [dcl.decl]. An *identifier* that names a coroutine parameter refers to the copy of the parameter [dcl.fct.def.coroutine].

[*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* or ~~*decltype-specifier*~~ *computed-type-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 [class.mfct.non.static]. — *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 [expr.prim.id.qual, dcl.type.simple, dcl.type.elab, dcl.mptr, namespace.udecl, temp.param, temp.names, temp.res]. — *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*~~ *computed-type-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*~~ *computed-type-specifier* nominates the type denoted by the ~~*decltype-specifier*~~ *computed-type-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 [temp.arg] of the corresponding *template-head* H [temp.mem], N nominates the primary template of T; H shall be equivalent to the *template-head* of T [temp.over.link].
- Otherwise, N nominates the partial specialization [temp.spec.partial] of T whose template argument list is equivalent to A [temp.over.link]; 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 form *nested-name-specifier* `template`$_{opt}$ `~` ~~*decltype-specifier*~~ *computed-type-specifier* nor of the form ~~*decltype-specifier*~~ *computed-type-specifier* `::` `~` *type-name*.

The result of a *qualified-id* Q is the entity it denotes [basic.lookup.qual]. 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 [dcl.struct.bind], 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-index-expression:
    id-expression `...` `[` *constant-expression* `]`

The *id-expression* P in a *pack-index-expression* shall be an *identifier* that denotes 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-expression* is a pack expansion ([temp.variadic]).

[*Note:* A *pack-index-expression* denotes the V^{th} element of the pack ([temp.variadic]). — *end note*]

� Unary operators [expr.unary.op]

[*Editor's note:* Modify [expr.unary.op]/p10]

The operand of the ~ operator shall have integral or unscoped enumeration type. Integral promotions are performed. The type of the result is the type of the promoted operand. Given the coefficients $\mathtt{x}_i$ of the base-2 representation[basic.fundamental] of the promoted operand `x`, the coefficient $\mathtt{r}_i$ of the base-2 representation of the result `r` is 1 if $\mathtt{x}_i$ is 0, and 0 otherwise. [*Note:* The result is the ones' complement of the operand (where operand and result are considered as unsigned). — *end note*] There is an ambiguity in the grammar when ~ is followed by a *type-name* or ~~*decltype-specifier*~~ *computed-type-specifier*. The ambiguity is resolved by treating ~ as the operator rather than as the start of an *unqualified-id* naming a destructor. [*Note:* Because the grammar does not permit an operator to follow the `.`, `->`, or `::` tokens, a ~ followed by a *type-name* or ~~*decltype-specifier*~~ *computed-type-specifier* in a member access expression or *qualified-id* is unambiguously parsed as a destructor name. — *end note*]

� Type names [dcl.name]

To specify type conversions explicitly, and as an argument of `sizeof`, `alignof`, `new`, or `typeid`, the name of a type shall be specified. This can be done with a *type-id*, which is syntactically a declaration for a variable or function of that type that omits the name of the entity.

type-id:
    type-specifier-seq abstract-declarator$_{opt}$

defining-type-id:
    defining-type-specifier-seq abstract-declarator$_{opt}$

abstract-declarator:
    ptr-abstract-declarator
    noptr-abstract-declarator$_{opt}$ *parameters-and-qualifiers trailing-return-type*
    abstract-pack-declarator

ptr-abstract-declarator:
    noptr-abstract-declarator
    ptr-operator ptr-abstract-declarator$_{opt}$

noptr-abstract-declarator:
  noptr-abstract-declarator$_{opt}$ *parameters-and-qualifiers*
  noptr-abstract-declarator$_{opt}$ `[` *constant-expression*$_{opt}$ `]` *attribute-specifier-seq*$_{opt}$
  `(` *ptr-abstract-declarator* `)`

abstract-pack-declarator:
  noptr-abstract-pack-declarator
  ptr-operator abstract-pack-declarator

noptr-abstract-pack-declarator:
  noptr-abstract-pack-declarator parameters-and-qualifiers
  ~~*noptr-abstract-pack-declarator* `[` *constant-expression*$_{opt}$ `]` *attribute-specifier-seq*$_{opt}$~~
  ...

[*Editor's note:* The sequence `...[constant-expression]` should always be treated as pack indexing. However we may want to allow `T(&...)[constant-expression]`, which is the object of CWG1488 [11]]

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

� 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*~~ *computed-type-specifier*
  placeholder-type-specifier
  nested-name-specifier$_{opt}$ *template-name*

computed-type-specifier:
  decltype-specifier
  pack-index-specifier

[...]

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

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

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]
decltype-specifier	the type as defined in [dcl.type.decltype]
pack-index-specifier	the type as defined in [dcl.type.pack.indexing]
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`"
...	

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-specifier* is a pack expansion ([temp.variadic]).

[*Note:* The *pack-index-specifier* denotes the type of the V^{th} element of the pack ([temp.variadic]). — *end note*]

� 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 [dcl.struct.bind], `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* [temp.param], `decltype(`E`)` is the type of the *template-parameter* after performing any necessary type deduction [dcl.spec.auto, dcl.type.class.deduct];
- otherwise, if E is an unparenthesized *id-expression* or an unparenthesized class member access [expr.ref], `decltype(`E`)` is the type of the entity named by E. If there is no such entity, the program is ill-formed;
- 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 [term.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])   x5;  // type is int
    decltype((pack...[0])) x6;  // 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* or ~~*decltype-specifier*~~ *computed-type-specifier* that denotes the destructor's class type. The invocation of a destructor is subject to the usual rules for member functions [class.mfct]; 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 $\ldots_{opt}$
  base-specifier-list , *base-specifier* $\ldots_{opt}$

base-specifier:

attribute-specifier-seq$_{opt}$ ~~*class-or-decltype*~~ *class-or-computed-type-specifier*

attribute-specifier-seq$_{opt}$ `virtual` *access-specifier*$_{opt}$ ~~*class-or-decltype*~~ *class-or-computed-type-specifier*

attribute-specifier-seq$_{opt}$ *access-specifier* `virtual`$_{opt}$ ~~*class-or-decltype*~~ *class-or-computed-type-specifier*

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

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

� Type equivalence [temp.type]

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-specifiers* refer to the same type only if their *constant-expressions* 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.

� 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 [dcl.fct]; the pattern is the *parameter-declaration* without the ellipsis.
- In a *using-declaration* [namespace.udecl]; the pattern is a *using-declarator*.
- In a template parameter pack that is a pack expansion [temp.param]:

    - 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* [dcl.init]; the pattern is an *initializer-clause*.
- In a *base-specifier-list* [class.derived]; the pattern is a *base-specifier*.
- In a *mem-initializer-list* [class.base.init] for a *mem-initializer* whose *mem-initializer-id* denotes a base class; the pattern is the *mem-initializer*.
- In a *template-argument-list* [temp.arg]; the pattern is a *template-argument*.
- In an *attribute-list* [dcl.attr.grammar]; the pattern is an *attribute*.
- In an *alignment-specifier* [dcl.align]; the pattern is the *alignment-specifier* without the ellipsis.
- In a *capture-list* [expr.prim.lambda.capture]; the pattern is the *capture* without the ellipsis.
- 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 *fold-expression* [expr.prim.fold]; 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 [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* $\mathtt{E}_K$.

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

� Dependent types [temp.dep.type]

[*Editor's note:* Add a bullet in paragraph 7]

A type is dependent if it is

- a template parameter,
- denoted by a dependent (qualified) name,
- a nested class or enumeration that is a direct member of a class that is the current instantiation,
- a cv-qualified type where the cv-unqualified type is dependent,
- a compound type constructed from any dependent type,
- an array type whose element type is dependent or whose bound (if any) is value-dependent,
- a function type whose parameters include one or more function parameter packs,
- a function type whose exception specification is value-dependent,
- denoted by a *simple-template-id* in which either the template name is a template parameter or any of the template arguments is a dependent type or an expression that is type-dependent or value-dependent or is a pack expansion,
- a *pack-index-specifier*, or
- denoted by `decltype(`*expression*`)`, where *expression* is type-dependent[temp.dep.expr].

� Type-dependent expressions [temp.dep.expr]

[*Editor's note:* Add a paragraph at the end of temp.dep.expr]

A *braced-init-list* is type-dependent if any element is type-dependent or is a pack expansion.

A *fold-expression* is type-dependent.

A *pack-index-expression* is type-dependent if its *id-expression* is type-dependent.

[...]

� 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* or a *pack-index-expression*.

- 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 [over.over], 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 [dcl.init.list] but the parameter does not have a type for which deduction from an initializer list is specified [temp.deduct.call]. [*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 declare a pack of function parameters. `T...[`n`]` is now a *pack-index-specifier*.
Rationale: Improve the handling of packs.
Effect on original feature: Valid C++23 code that declares a pack of parameter without specifying a `declarator-id` becomes 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 extend our appreciation to Sean Baxter for his work on Circle and to Barry Revzin for his work on P1858R2 [8], both works being the foundation of the design presented here.

Thanks also to Lewis Baker for his valuable feedback on this paper, Nina Dinka Ranns for her help with wording, and Lori Hughes for editing this paper.

References

[1] Gašper Ažman, Mateusz Pusz, Colin MacLean, and Bengt Gustafsonn. P1985R1: Universal template parameters. https://wg21.link/p1985r1, 5 2020.

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

[3] Corentin Jabot, Pablo Halpern, John Lakos, Alisdair Meredith, Joshua Berne, and Gašper Ažman. P2632R0: A plan for better template meta programming facilities in c++26. https://wg21.link/p2632r0, 10 2022.

[4] Zach Laine. P2726R0: Better std::tuple indexing. https://wg21.link/p2726r0, 11 2022.

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

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

[7] Barry Revzin. P1858R0: Generalized pack declaration and usage. https://wg21.link/p1858r0, 10 2019.

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

[9] Barry Revzin. P2671R0: Syntax choices for generalized pack declaration and usage. https://wg21.link/p2671r0, 10 2022.

[10] Barry Revzin and Jonathan Wakely. P1061R5: Structured bindings can introduce a pack. https://wg21.link/p1061r5, 5 2023.

[11] Richard Smith. CWG1488: abstract-pack-declarators in type-ids. https://wg21.link/cwg1488, 3 2012.

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

[13] Matthew Woehlke. P0311R0: A unified vision for manipulating tuple-like objects. https://wg21.link/p0311r0, 3 2016.

[14] Matthew Woehlke. P0535R0: Generalized unpacking and parameter pack slicing. https://wg21.link/p0535r0, 2 2017.

[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