Library Support for Expansion Statements

Document #:	P1789R1
Date:	2025-04-15
Project:	Programming Language C++
Audience:	Library Evolution

Reply-to:	Alisdair Meredith <ameredith1@bloomberg.net> Jeremy Rifkin <jeremy@rifkin.dev> Matthias Wippich <mfwippich@gmail.com>

Contents

Abstract

This paper proposes support for using `std::integer_sequence` in structured bindings and expansion statements.

Revision history

R1 April 2025 (mid-term mailing)

— Updated to use the new expansion statement syntax from [P1306R3]
— Added section on use in structured bindings.

R0 June 2019 (pre-Cologne mailing)

Original version of the paper for the 2019 pre-Cologne mailing.

1 Introduction

Packs from structured bindings ([P1061R10]) and expansion statements ([P1306R3]) are new language features slated for C++26. These features both allow compile-time repetition of destructurable types through either fold expressions or a for-loop-like syntax.

However there are no standard compile-time sequences of integers that support these new language features. Such sequences are integral to supporting indexed access when iterating or folding over multiple sequences.

A simple fix would be to enhance the library template `std::integer_sequence` implementing the structured bindings protocol, enabling its use with both new features.

1.1 History

Expansion statements are a new language feature originally intended for C++20 that will add compile-time iteration and code generation to C++26.

Despite feature approval by EWG for C++20, it failed to land in time due to design and implementability concerns raised during the Core wording review. Those concerns have now been addressed and [P1306R3] was once again approved by EWG for adoption, now targeting C++26.

The first revision of this paper was written in 2019. We are revisiting it now that expansion statements are landing in C++26 as an essential component of reflection, and packs from structured bindings have been introduced making this proposal even more useful.

1.2 An essential C++26 feature

C++26 introduces two new language features that facilitate compile-time repetition over packs. We view the ability to use `std::integer_sequence` with these features as essential to their success.

For example, a common pattern for obtaining a pack of integers of length `COUNT` within a function template is to take advantage of lambda template arguments and type deduction on the function-call argument to the lambda to obtain an integer sequence pack, moving the majority of the function template's logic into the lambda itself.

```
[]<std::size_t ...INDEX>(std::index_sequence<INDEX...>) {
  // here lies the enclosing function's implementation
} (std::make_index_sequence<COUNT>());
```

Aside from the roundabout and cumbersome syntax, this introduces unnecessary nesting in code already struggling under the weight of template syntax. A cursory GitHub Code Search finds 3.3k instances this pattern using lambdas to produce integer packs. While this is not an every-day language construct, it is not an uncommon pattern and important within its domain. We expect more C++ metaprogramming in the coming years so we need a more ergonomic solution.

This paper proposes adding support for the structured bindings protocol to `std::integer_sequence` to enable refactorings that use `std::integer_sequence` in an expansion statement or structured binding to produce more readable and maintainable code. Embracing the new language features provided by C++26 we would rewrite code that used the unnecessary lambda pattern as either

```
constexpr auto [...INDEX] = std::make_index_sequence<COUNT>();
// the parameter pack `INDEX` is now usable directly in the function body
// ...
```

or

```
template for(constexpr size_t INDEX : std::make_index_sequence<COUNT>()) {
  // ...
}
```

2 Motivation and Use Cases

Expansion statements allow for iteration over a variety of compile-time sequences such as parameter packs and (`constexpr`) spans of reflections — see [P2996] for examples of reflection. In particular, expansion statements can iterate and generate code for any type that can, during constant evaluation, be destructured though the tuple protocol. Additionally, packs can now be introduced with structured bindings and used with fold expressions.

2.1 Comparing tuples

It is relatively straightforward to implement comparison for `std::tuple` using a fold expression, but needs either a lambda expression or an auxiliary function to introduce a context for the pack expansion.

```
template<class ...TTypes, class ...UTypes>
   requires (sizeof...(TTypes) == sizeof...(UTypes))
constexpr
bool operator==(tuple<TTypes...> const & lhs, tuple<UTypes...> const & rhs) {
   auto impl = [&, this]<size_t ...INDEX>(index_sequence<INDEX...>) {
      return ((get<INDEX>(lhs) == get<INDEX>(rhs)) && ...);
   };

   return impl(index_sequence_for<TTypes...>{});
}
```

Note that this is the more readable form of this code, often developers who are overly familiar with this idiom will eliminate the local variable, producing code that is opaque to all but the most seasoned developer.

```
template<class ...TTypes, class ...UTypes>
   requires (sizeof...(TTypes) == sizeof...(UTypes))
constexpr
bool operator==(tuple<TTypes...> const & lhs, tuple<UTypes...> const & rhs) {
   return [&, this]<size_t ...INDEX>(index_sequence<INDEX...>) {
      return ((get<INDEX>(lhs) == get<INDEX>(rhs)) && ...);
   }(index_sequence_for<TTypes...>());
}
```

It is straightforward to implement comparison using a fold expression when `index_sequence` is destructurable.

```
template<class ...TTypes, class ...UTypes>
   requires (sizeof...(TTypes) == sizeof...(UTypes))
constexpr
bool operator==(tuple<TTypes...> const & lhs, tuple<UTypes...> const & rhs) {
   constexpr auto [...INDEX] = index_sequence_for<TTypes...>{};
   return ((get<INDEX>(lhs) == get<INDEX>(rhs)) && ...);
}
```

2.2 Implementing `std::tuple::swap`

It is similarly straightforward to implement `tuple::swap` using a fold expression over the comma operator, but also somewhat of a hack. This is the kind of code we would like to be able to write more cleanly using an expansion statement.

```
template <class ...TTypes>
constexpr
void tuple<TTypes...>::swap(tuple& other) noexcept((is_nothrow_swappable_v<TTypes> and ...))
{
   auto impl = [&, this]<size_t ...INDEX>(index_sequence<INDEX...>) {
      ((void)swap(get<INDEX>(*this), get<INDEX>(other)), ...);
   };

   impl(index_sequence_for<TTypes...>{});
}
```

Note that in addition to the internal use of a lambda expression to create the parameter pack to fold over, we must also cast the result of the `swap` call to `void` in case users provide an ADL-discoverable `swap` function that returns a user defined type that, in turn, provides an overload for the comma operator.

We can eliminate the fold expression and stop worrying about the surprising corner cases handling ADL-`swap` by using a C++26 expansion statement like so:

```
template <class ...TTypes>
constexpr
void tuple<TTypes...>::swap(tuple& other) noexcept((is_nothrow_swappable_v<TTypes> and ...))
{
   auto impl = [&, this]<size_t ...INDEX>(index_sequence<INDEX...>) {
      template for(constexpr size_t N : {INDEX...}) {
        swap(get<N>(*this), get<N>(other));
      }
   };

   impl(index_sequence_for<TTypes...>{});
}
```

However, there is no easy way to eliminate the lambda expression, as we cannot iterate over an `integer_sequence` using just the facilities provided in [P1306R3]. In fact, this "new and improved" version is actually longer than the single fold expression that it replaces, although we claim that the code is easier to read with fewer subtleties that readers must be aware of.

The heart of the problem is that expansion statements are an excellent tool to iterate over a single sequence, but do not provide an index to support iterating over two sequences in parallel. We propose that the simplest way to resolve the concerns is to add the missing pieces that would enable use of `integer_sequence` in a structured binding. That is sufficient to support use in expansion statements, and is general enough to be a feature in its own right. With such support, the `tuple::swap` example simplifies to:

```
template <class ...TTypes>
constexpr
void tuple<TTypes...>::swap(tuple& other) noexcept((is_nothrow_swappable_v<TTypes> and ...))
{
   template for(constexpr size_t INDEX : index_sequence_for<TTypes...>{}) {
      swap(get<INDEX>(*this), get<INDEX>(other));
   }
}
```

2.3 Splicing reflections with structured bindings

Supporting decomposition of `integer_sequence`s interacts well with the proposed changes of [P1061R10] Structured Bindings can introduce a Pack and [P2686R5] `constexpr` structured bindings.

In a lot of existing code the following pattern is used to introduce a sequence of integers as a pack of constants.

```
[]<std::size_t ...INDEX>(std::index_sequence<INDEX...>) {
  // ...
} (std::make_index_sequence<COUNT>());
```

Not only is this rather verbose, it introduces a new function scope. This can be avoided by introducing the pack of integers `INDEX` through a structured binding instead.

```
constexpr auto [...INDEX] = std::make_index_sequence<COUNT>();
```

As [P1306R3] mentions, introducing a new function scope can be problematic. For instance, reflections of function parameters ([P3096R5]) can be spliced only within their corresponding function body. Expansion statements alleviate this issue, but are not usable whenever we want to fold over a pack of reflections or expand them into an argument list.

For instance, consider the following (invalid) example:

```
void foo(int x, char c) {
   [:expand(parameters_of(^^foo)):]
   >> []<auto ...parameters>(){
      bar([:parameters:]...); // oops
   }
}
```

By using the features introduced in [P1061R10] and [P2686R5], the introduction of a new function scope can be avoided.

```
void foo(int x, char c) {
   constexpr auto [...INDEX] = std::make_index_sequence<parameters_of(^^foo).size()>();
   bar([:parameters_of(^^foo)[INDEX]:]...);
}
```

3 Proposed Solution

3.1 Make `integer_sequence` support structured bindings

`integer_sequence` is missing three things in order to support use in structured bindings:

— A partial specialization for `tuple_size`
— A partial specialization for `tuple_elememt`
— Overloads of `get<INDEX>()`

The first two bullets are fairly straightforward to implement. For the `get` function, we propose a single overload taking an `integer_sequence` by value, as it is an immutable empty type, and likewise returning its result by value.

3.2 Sample implementation

```
template<class T, T ...VALUES>
struct tuple_size<integer_sequence<T, VALUES...>>
     : integral_constant<size_t, sizeof...(VALUES)>
{ };

template<size_t INDEX, class T, T ...VALUES>
   requires (INDEX < sizeof...(VALUES))
struct tuple_element<INDEX, integer_sequence<T, VALUES...>> {
  using type = T;
};

template<size_t INDEX, class T, T ...VALUES>
   requires (INDEX < sizeof...(VALUES))
constexpr T get(integer_sequence<T, VALUES...>) noexcept {
   return VALUES...[INDEX];
}
```

4 Wording

Make the following changes to the C++ Working Draft. All wording is relative to [N5008], the latest draft at the time of writing.

22.2.1 [utility.syn] Header <utility> synopsis

1 The header <utility> contains some basic function and class templates that are used throughout the rest of the library.

```
#include <compare>          // see 17.12.1
#include <initializer_list> // see 17.11.2

namespace std {
  // 22.2.2, swap
  ...

  // 21.2, Compile-time integer sequences
  template<class T, T...>
    struct integer_sequence;
  template<size_t... I>
    using index_sequence = integer_sequence<size_t, I...>;

  template<class T, T N>
    using make_integer_sequence = integer_sequence<T, see below>;
  template<size_t N>
    using make_index_sequence = make_integer_sequence<size_t, N>;

  template<class... T>
    using index_sequence_for = make_index_sequence<sizeof...(T)>;
```

```
  // forward declaration for structured binding support
  template<class T> struct tuple_size;
  template<size_t I, class T> struct tuple_element;

  // structured binding support for integer_sequence
  template<class T, T... Values>
    struct tuple_size<integer_sequence<T, Values...>>;
  template<size_t I, class T, T... Values>
    struct tuple_element<I, integer_sequence<T, Values...>>;

  template<size_t I, class T, T... Values>
  constexpr T get(integer_sequence<T, Values...>) noexcept;
```

```
  // 22.3, class template pair
  template<class T1, class T2>
    struct pair;

  ...

  // 22.3.4, tuple-like access to pair
  ~~template struct tuple_size;~~
  ~~template< size_t I, classT> struct tuple_element;~~

  template<class T1, class T2> struct tuple_size<pair<T1, T2>>;
```

```
  template<size_t I, class T1, class T2> struct tuple_element<I, pair<T1, T2>>;

}
```

21.2 [intseq] Compile-time integer sequences

21.2.4 [intseq.binding] Structured binding support

```
template<class T, T... Values>
  struct tuple_size<integer_sequence<T, Values...>>
    : integral_constant<size_t, sizeof...(Values)> { };

template<size_t I, class T, T... Values>
  struct tuple_element<I, integer_sequence<T, Values...>> {
    using type = T;
  };
```

1 *Mandates:* `I < sizeof...(Values)`.

```
template<size_t I, class T, T... Values>
constexpr T get(integer_sequence<T, Values...>) noexcept;
```

3 *Mandates:* `I < sizeof...(Values)`.

4 *Returns:* `Values...[I]`.

5 Acknowledgements

Thanks to Michael Park for the pandoc-based framework used to transform this document's source from Markdown.

Thanks to Vittorio Romeo and Daveed Vandevoorde for their insights into the initial contents of this paper.

6 References

[N5008] Thomas Köppe. Working Draft, Programming Languages — C++.
https://wg21.link/n5008

[P1061R10] Barry Revzin, Jonathan Wakely. 2024-11-24. Structured Bindings can introduce a Pack.
https://wg21.link/p1061r10

[P1306R3] Dan Katz, Andrew Sutton, Sam Goodrick, Daveed Vandevoorde. 2024-10-14. Expansion statements.

https://wg21.link/p1306r3

[P2686R5] Corentin Jabot, Brian Bi. 2024-11-12. constexpr structured bindings and references to constexpr variables.
https://wg21.link/p2686r5

[P2996] Wyatt Childers, Peter Dimov, Dan Katz, Barry Revzin, Andrew Sutton, Faisal Vali, Daveed Vandevoorde. Reflection for C++26.
https://wg21.link/p2996

[P3096R5] Adam Lach, Walter Genovese. 2024-12-14. Function Parameter Reflection in Reflection for C++26.
https://wg21.link/p3096r5