Expansion statements

Version history

r2 Adoption of `template for` syntax. Added support for init-statement, folded pack expansion into new *expansion-init-list* mechanism. Updated reflection code to match P2996. Minor updates to wording: updated handling of `switch` statements, work around lack of general non-transient `constexpr` allocation, eliminated need for definition of an "intervening statement", rebased onto working draft, updated feature macro value, fixed typos. Addressed CWG review feedback.

r1 Adopted a unified syntax for different forms of expansion statements. Further refinement of semantics to ensure expansion can be supported for all traversable sequences, including ranges of input iterators. Added discussion about `break` and `continue` within expansions.

r0 Superseded and extended P0589R0, ("Tuple-based for-loops"). to work with more destructurable objects (e.g., classes, parameter packs). Added a separate constexpr-for variant that a) makes the loop variable a constant expression in each repeated expansion, and b) makes it possible to expand constexpr ranges. The latter feature is particularly important for static reflection.

Introduction

This paper proposes a new kind of statement that enables the compile-time repetition of a *statement* for each element of a tuple, array, class, range, or brace-delimited list of expressions. Existing methods for iterating over a heterogeneous container inevitably leverage recursively instantiated templates, which allow some part of the repeated statement to vary (e.g., by type or constant) in each instantiation.

While such behavior can be encapsulated in a single library operation (e.g., Boost.Hana's `for_each`) or, potentially in the future, using the `[:expand(...):]` construct built on top of P2996 reflection facilities, there are several reasons to prefer language support. First, repetition is a fundamental building block of algorithms, and should be expressible directly without complex template instantiation strategies.

Second, such repetition should be as inexpensive as possible. Recursively instantiating templates generates a large number of specializations, which can consume significant compiler memory and compile time. Thirdly, library-based approaches rely on placing the repeated statements in a lambda body, which changes the semantics of something like a `return` statement. Lastly, "iteration" over destructurable classes effectively requires language support to implement correctly.

History of this proposal

The idea for expansion statements began with Andrew Sutton's 2017 proposal for tuple-based `for` loops ([P0589](#)), which noted iteration over parameter packs and *braced-init-list*s as directions for future work.

P1306R0 superseded Andrew's initial proposal in Aug 2018, adding support for iteration over packs (but not iteration over *braced-init-list*s). The R1 revision was accepted by EWG in Kona 2019:

Strongly favor: 22 — Favor: 18 — Neutral: 2 — Against: 0 — Strongly against: 2

A subsequent EWG vote in Cologne 2019 affirmed support for the `template for` syntax over several proposed alternatives:

Strongly favor: 3 — Favor: 19 — Neutral: 2 — Against: 3 — Strongly against: 1

Removal of pack expansion

Review by CWG surfaced ambiguity in the syntax proposed for pack expansion. Consider a function:

```
template <typename... Ts> void fn(Ts... vs) {
  ([&](auto p) {
    template for (auto &v : vs) { /* ... */ }
  }(vs), ...);
}
```

and one of its call sites:

```
fn(array {1,2,3,4}, array {1,3,5,7}, array {2,4,6,8});
```

It is far from clear whether the expansion statement containing `vs` expands over:
- each of the three `array` arguments, once for each invocation of the lambda, or
- each of the four `int` elements of a different `array` argument for each invocation of the lambda

In response to such ambiguity, support for pack iteration was dropped from this proposal in July 2019.

This proposal was last reviewed by CWG in a Jan 2022 telecon; the draft reviewed at that time can be found [here](#). The current revision attempts to address feedback received at that time.

Revisiting pack expansion in 2024

In the present draft, we propose a syntax extension that permits iteration over arbitrary lists of expressions (akin to a *braced-init-list*) which covers iteration over packs as a special case, thus providing the full power of expansion statements as first envisioned by P0589. The proposed syntax polled favorably by EWG during discussions of P2994 ("On the Naming of Packs", by Barry Revzin) in Tokyo 2024, and we believe it steers clear of any ambiguity.

Basic usage

Here is an example demonstrating how the Boost.Hana library iterates over the elements of a tuple:

```
auto tup = std::make_tuple(0, 'a', 3.14);
hana::for_each(tup, [&](auto elem) {
  std::cout << elem << std::endl;
});
```

The `for_each` function applies the generic lambda to each element of the tuple, printing them in turn. Each call instantiates a new function containing a call to `cout` for the corresponding tuple element.

Using the feature described by this proposal, that code could instead be written as:

```
auto tup = std::make_tuple(0, 'a', 3.14);
template for (auto elem : tup)
  std::cout << elem << std::endl;
```

The `template for` statement expands the body of the loop once for each element of the tuple. In other words, the expansion statement above is equivalent to the following:

```
auto tup = std::make_tuple(0, 'a', 3.14);
{
  auto elem = std::get<0>(tup);
  std::cout << elem << std::endl;
}
{
  auto elem = std::get<1>(tup);
  std::cout << elem << std::endl;
}
{
  auto elem = std::get<2>(tup);
  std::cout << elem << std::endl;
}
```

The *expansion-init-list* syntax proposed as of this paper's R2 revision allows the above code to be written even more concisely:

```
template for (auto elem : {0, 'a', 3.14})
  std::cout << elem << std::endl;
```

Although this looks like a *braced-init-list*, the similarity is purely aesthetic: there is no list-initialization taking place; the braces serve only to demarcate the domain of expansion.

An expansion statement *is not a loop*: it is a sequence of instantiations of a provided statement, in which the associated variable is initialized to each successive element in the tuple. Because the variable is re-declared in each instantiation of the body, its type is allowed to vary. This makes expansion statements a useful tool for defining a number of algorithms on heterogeneous collections.

An expansion statement allows expansion over the following:
- Destructurable classes (including plain structs and tuples).
- Constexpr ranges (including compile-time vectors)
- Brace-delimited lists of expressions ("*expansion-init-list*s", including pack expansions)

Expansion and static reflection

The ability to repeat statements for collections of entities is central to practically all useful reflection algorithms. Here is an early generic implementation of Howard Hinnant's *Types Don't Know #* proposal (N3980) using the facilities from P2996.

```
template<HashAlgorithm H, StandardLayoutType T>
bool hash_append(H& algo, const T& t) {
  template for (constexpr auto member : nonstatic_data_members_of(^T))
    hash_append(h, t.[:member:]);
}
```

Since `constexpr` appears as a *decl-specifier* of the variable `member`, that variable is usable in constant expressions in each expansion (e.g., suitable for use in a template argument list). This is necessary in order to "splice" the reflection variable using the `[:member:]` syntax, which yields a resolved reference to the corresponding data member (note: this works even if the type `T` contains bitfield members).

This all works because `nonstatic_data_members_of` returns a forward-traversable sequence of values whose extent is known during compilation (in this case, a compile-time `std::vector`). One subtlety worth noting is that defining a `constexpr std::vector` (i.e., to construct the "domain" of expansion) requires us to adopt tactics similar to those deployed for P3032 ("*Less transient constexpr allocation*"): We carve out a special provision from [expr.const] allowing compile-time use of `operator`

`new` and `operator delete` within an *expansion-initializer,* with the caveat that the allocated memory must be deallocated within the evaluation of the expansion statement.

The fully expanded statement is roughly equivalent to the following:

```
{
  constexpr std::vector members = nonstatic_data_members_of(^T);
    // ^ validity requires special provision from [expr.const]
  {
    constexpr member0 = *std::next(std::begin(members), 0);
    hash_append(h, t.[:member0:]);
  }
  {
    constexpr member1  *std::next(std::begin(members), 1);
    hash_append(h, t.[:member1:]);
  }
...
  {
    constexpr memberK = *std::next(std::begin(members), K);
    hash_append(h, t.[:memberK:]);
  }
}
```

A total of *K*-many instances of the body are expanded, where *K* is:

```
std::distance(std::begin(members), std::end(members)).
```

Note that expansion only occurs when the range is non-dependent (e.g., during template instantiation).

Without expansion statements, something akin to the `expand` workaround described in P2996 ("Reflection for C++26") is needed to traverse a list of reflections. One such implementation is shown below:

P2996 code:
```
namespace __impl {  // start 'expand' definition
  template<auto... vals>
  struct replicator_type {
    template<typename F>
      constexpr void operator>>(F body) const {
        (body.template operator()<vals>(), ...);
      }
  };

  template<auto... vals>
```

```
    replicator_type<vals...> replicator = {};
  }  // namespace __impl

  template<typename R>
  consteval auto expand(R range) {
    std::vector<std::meta::info> args;
    for (auto r : range) {
      args.push_back(std::meta::reflect_value(r));
    }
    return substitute(^__impl::replicator, args);
  }  // end 'expand' definition

  template<typename T>
  bool hash_append(const T& t) {
    [:expand(nonstatic_data_members_of(^T)):] >> [&t]<auto Member> {
      hash_append(t.[:Member:]);
    };
  }
```

Break and continue

At this time, we propose disallowing `break` and `continue` within expansion statements. Their meaning can be defined and implemented at a later time, but our concern is that users will confuse these statements as providing a means of short-circuiting the expansion itself (which they would not).

Syntax and semantics

The syntax for an expansion statement is similar to that of a range-based for loop.

expansion-statement:
    `template for (`*init-statement$_{opt}$ for-range-declaration* `:` *expansion-initializer*`)* statement*

An *expansion-statement* expands statically to a statement equivalent to the following pattern, with the caveat that the `__range` variable is elided when the *expansion-initializer* is an *expansion-init-list*.

```
  {
```
    init-statement

    constexpr-specifier$_{opt}$ `auto __range =` *expansion-initializer*`;`
    constexpr-specifier$_{opt}$ `auto __begin =` *begin-expr*`;`
    constexpr-specifier$_{opt}$ `auto __end =` *end-expr*`;`

```
constexpr auto __iter_0 = __begin;
<stop expansion if __iter_0 == __end>
{
  for-range-declaration = get-expr(__iter_0)>;
  statement
}
constexpr auto __iter_1 = next-expr(__iter_0);
<stop expansion if __iter_1 == __end>
{
  for-range-declaration = get-expr(__iter_1)>;
  statement
}
constexpr auto __iter_2 = next-expr(__iter_1);

// … repeats until __iter_K == __end
}
```

The optional *constexpr-specifier* is the token `constexpr` only if the *expansion-declaration* includes `constexpr` in its *decl-specifier-seq*; it is otherwise empty. The meaning of the placeholder expressions *begin-expr*, *end-expr*, *get-expr*, *next-expr* depend on the form of *expansion-initializer*. There are three such forms that the *expansion-initializer* may take.

If the *expansion-initializer* is of the form `{ expr`$_0$`, ..., expr`$_K$` }`, the expansion is performed over an integer index `I` into the *K* expressions enclosed by the braces (note: some or all of the expressions may be from a pack expansion), and the placeholder expressions are:
- *begin-expr* is `0u`
- *end-expr* is `K`
- *get-expr(*`I`*)* is `expr`$_I$
- *get-expr(*`I`*)* is `expr`$_{I+1}$.

Otherwise, if the substitution of the *expansion-initializer* into a range-based `for` statement of the form
        `for (auto&& __unspecified :` *expansion-initializer*`) ;`
would succeed, the expansion is performed over a sequence of iterators `I` ranging over *expansion-initializer*, and the placeholder expressions are:
- *begin-expr* and *end-expr* are the same as for a range-based for loop
- *get-expr(*`I`*)* is `*I`
- *next-expr(*`I`*)* is `std::next(I)`

Otherwise, if the substitution of the *expansion-initializer* into a structured binding of the form
        `auto [I0, I1, ..., IK] =` *expansion-initializer*
would succeed, the expansion is performed over an integer index `I` into the sequence of members selected for destructuring, and the placeholder expressions are:
- *begin-expr* is `0u`

- *end-expr* is K
- *get-expr(*I*)* is the Ith entity named by the structured binding
- *next-expr(*I*)* is I + 1

Note that the range form of the expansion is intended to be valid for any iterable entity whose extent is statically known. In the most general case, this emulates the hand-unrolling of a range-based `for` loop over an input range (i.e., a range with input iterators). Here, care must be taken not to "accidentally" consume range elements by calling `std::distance` or advancing multiple elements in a single call to `std::next`. For destructurable objects and brace-delimited lists of expressions, the expansion can be trivially implemented in terms of a simple integer index. A compiler might also optimize (for compile-time) certain range-based expansions if it can determine the iterator category of the range.

Examples:

```
auto tup = std::make_tuple(0, 'a');
template for (auto& elem : tup)
  elem += 1;
assert(tup == make_tuple(1, 'b'));
```

A possible expansion is:

```
{
  auto &&__range = tup;
  {
    auto& elem = std::get<0>(__range);
    elem += 1;
  }
  {
    auto& elem = std::get<1>(__range);
    elem += 1;
  }
}
```

Below is an example of a `constexpr` expansion (valid thanks to proposed changes to [expr.const]):

```
template for (constexpr int n : std::vector {1, 2, 3})
  f<n>();
```

… and its expansion:

```
{
  constexpr auto __range = std::vector {1, 2, 3};
  constexpr auto __end = __range.end();
```

```
    constexpr auto __iter_0 = __range.begin();
    {
      constexpr int n = *__iter_0;
      f<n>();
    }
    constexpr auto iter_1 = std::next(__iter_0);
    {
      constexpr int n = *__iter_1;
      f<n>();
    }
    constexpr auto iter_2 = std::next(__iter_1);
    {
      constexpr int n = *__iter_2;
      f<n>();
    }
  }
```

Observations and notes

In the following subsections we discuss some specification details and implementation notes.

Required header files

As with the range-based `for` loop, no additional header files are required to use this feature. Many expansions (e.g., over arrays) are defined using only core language constructs and do not require header files. Expanding over tuples does require the `<tuple>` header file, but that will almost certainly have been included before the use of the first *expansion-statement* over such an object.

Enumerating loop bodies

It may be useful to access the instantiation count in the loop body. This could be achieved by using an `enumerate` facility:

```
  template for (auto x : enumerate(some_tuple)) {
    // x has a count and a value
    std::println("{}: {}", x.count, x.value);

    // The count is also a compile-time constant.
    using T = decltype(x);
    std::array<int, T::count> a;
  }
```

The `enumerate` facility returns a simple tuple adaptor whose elements are (`count`, `value`)-pairs, and should be fairly straightforward to implement.

Implementation experience

A historical fork of Clang 8.0.0 implemented expansion over destructurable classes and over constant ranges. Expansion over *expansion-init-list*s, as proposed by this paper, is yet to be implemented.

For these *expansion-statement*s to work, the enclosed *statement* must be parsed as if inside a template and then repeatedly instantiated. Moreover, names appearing in expressions within the enclosed *statement* may be not ODR-used, even in a non-dependent context: if the *expansion-initializer* produces no elements, the result of expansion is an empty statement, and all statements, expressions, and declarations within the enclosed *statement* will be effectively erased from the program.

Related discussion

Apparently, there was an overlooked discussion about this feature on std.proposals in 2013 (https://groups.google.com/a/isocpp.org/forum/#!topic/std-proposals/vseNksuBviI).

Suggested Wording

6.4.2 Point of declaration [basic.scope.pdecl]

Add the following:

> 11. The locus of a *for-range-declaration* of a range-based `for` statement (_stmt.ranged_) or expansion statement (_stmt.expand_) is immediately after the *for-range-initializer* or *expansion-initializer*.

6.4.3 Block scope [basic.scope.block]

Update (1.1) to include expansion statements:

> (1.1) — selection, or iteration, or expansion statement (_stmt.select_, _stmt.iter_, _stmt.expand_)

6.7.7 Temporary objects [class.temporary]

Update (7) to extend the lifetime of temporaries created by an expansion-initializer.

7. The fourth context is when a temporary object other than a function parameter object is created in the *for-range-initializer* of a range-based `for` statement, or in an *expansion-initializer* of an expansion statement. If such a temporary object would otherwise be destroyed at the end of the *for-range-initializer* or *expansion-initializer* full-expression, the object persists for the lifetime of the reference initialized by the *for-range-initializer* or for the evaluation of the *expansion-statement*.

7.7 Constant expressions [expr.const]

Update paragraph 5 as follows (assuming changes from [P3032](#)):

5. An expression E is a *core constant expression* unless the evaluation of E, following the rules of the abstract machine (_intro.execution_), would evaluate one of the following:

…

   (5.18) — a *new expression* (_expr.new_), unless the selected allocation function is a replaceable global allocation function (_new.delete.single_, _new.delete.array_) and either E is in an immediate function context, E is the *expansion-initializer* of an *expansion-statement*, or the allocated storage is deallocated within the evaluation of E;

…

   (5.19) — a *delete expression* (_expr.delete_), unless it deallocates a region of storage allocated within the evaluation of E, or within the evaluation of an *expansion-statement* for which E is the *expansion-initializer*.

8.1 Preamble [stmt.pre]

Add the following:

1. Except as indicated, statements are executed in sequence.
   statement:
   >*labeled-statement*
   >*attribute-specifier-seq$_{opt}$ expression-statement*
   >*attribute-specifier-seq$_{opt}$ compound-statement*
   >*attribute-specifier-seq$_{opt}$ selection-statement*
   >*attribute-specifier-seq$_{opt}$ iteration-statement*
   >*attribute-specifier-seq$_{opt}$ expansion-statement*
   >*attribute-specifier-seq$_{opt}$ jump-statement*
   >*declaration-statement*
   >*attribute-specifier-seq$_{opt}$ try-block*

2. A substatement of a statement is one of the following:
   (2.1) — for a *labeled-statement*, its contained *statement*,
   (2.2) — for a *compound-statement*, any statement of its *statement-seq*,

(2.3) — for a *selection-statement*, any of its *statement*s (but not its *init-statement*), ~~or~~

(2.4) — for an *iteration-statement*, its contained *statement* (but not an *init-statement*)~~:~~, or

(2.5) — for an *expansion-statement*, its contained *statement (*but not an *init-statement).*

…

3. A *statement* S1 *encloses* a *statement* S2 if

(3.1) — S1 is a substatement of S1,

(3.2) — S1 is a *selection-statement* ~~or~~, *iteration-statement*, or *expansion-statement* and S2 is the *init-statement* of S1,

…

8.2 Labeled statement [stmt.label]

Add the following:

(3.2) — a label declared in S shall only be referred to by a statement (_stmt.goto_) in S.

4. An identifier label shall not occur in an *expansion-statement* (_stmt.expand_).

8.6.5 The range-based `for` statement [stmt.ranged]

Add the following paragraph.

3. An expression is *iterable* if, when the expression is treated as a *for-range-initializer* (_stmt.iter.general_), expressions `begin-expr` and `end-expr` can be determined as specified above and if they are of the form `begin(range)` and `end(range)`, argument-dependent lookup finds at least one function or function template for each.

8.7 Expansion statements [stmt.expand]

Insert this section after [stmt.iter] (and renumber accordingly).

1. Expansion statements specify compile-time repetition, with substitutions, of their substatement.

expansion-statement:
      `template for (` *init-statement*$_{opt}$ *for-range-declaration* `:` *expansion-initializer* `)` *statement*

expansion-init-list:
      `{` *expression-list* `}`

expansion-initializer:
      expression
      expansion-init-list

[*Note 1: An init-statement ends with a semicolon. — end note*]

2. The contained *statement* of an *expansion-statement* is a control-flow-limited statement (_stmt.label_).

3. In the *decl-specifier-seq* of a *for-range-declaration* in an *expansion-statement*, each *decl-specifier* shall be either a *type-specifier* or `constexpr`.

4. The substatement is implicitly parameterized by a non-type template parameter of type `int` that is used to form the *initializer* for the *for-range-declaration*. If the *expansion-initializer* is iterable (_stmt.ranged_), the type of the *initializer* for the *for-range declaration* is `decltype(*I)` where `I` is an iterator into the range. Otherwise, if the *expansion-initializer* is not an *expansion-init-list*, then it shall be a destructurable expression (_dcl.struct.bind_) and the *initializer* is type-dependent (_stmt.dep.expr_). [*Note:* The name declared by a *for-range-declaration* for an *expansion-statement* is value-dependent (_temp.dep.constexpr_) if the *expansion-initializer* is iterable, and is otherwise type-dependent (_temp.dep.expr_) if declared with a placeholder type. -- *end note*]

5. For the purpose of name lookup and instantiation, the *for-range-declaration* and the contained *statement* of the *expansion-statement* are together considered a template definition.

6. An *expansion-statement* is *expanded* (as described below) if its *expansion-initializer* neither contains a type-dependent expression nor is itself a value-dependent iterable expression. Expansion entails the repetition of a statement for each element of the *expansion-initializer*. Each repetition is called an *expansion* and is an instantiation (_temp.spec_) of the *for-range-declaration* (including its implied initialization) together with the *statement*.

7. If the *expansion-initializer* is an *expansion-init-list*, the *expansion-statement* is expanded once for each expression in the contained *expression-list*; the expansion is equivalent to:

```
{
    init-statement
    {   // i^th repetition of the substatement
        for-range-declaration = get-expr_i ;
        statement
    }
}
```

where $get\text{-}expr_i$ is the ith *expression* in the *expression-list* of the *expansion-init-list*.

8. Otherwise if the *expansion-initializer* is an iterable expression (_stmt.ranged_) and the *expansion-initializer* does not have array type, the *expansion-statement* is expanded once for each

element in the range computed by the *expansion-initializer*; the expansion is equivalent to:

```
{
    init-statement
    constexpr auto range = expansion-initializer ;
    {  //  iᵗʰ repetition of the substatement
        for-range-declaration = * get-exprᵢ ;
        statement
    }
}
```

where *get-expr$_i$* is a prvalue for the ith iterator in the sequence yielded by the evaluation of the following `for` loop in the context of a manifestly constant-evaluated expression (_expr.const_):

```
for (auto iter = begin-expr ; iter != end-expr ; ++iter)
    /* yield */ iter ;
```

The names *begin-expr* and *end-expr* are the expressions determined using the rules for range-based for statements (_stmt.ranged_). The names `range` and `iter` are for exposition only.

9. Otherwise, the *expansion-initializer* shall be destructurable, and the *expansion-statement* is expanded once for each element of the *identifier-list* of a structured binding declaration of the form `auto&& [u₁, u₂, …, uₙ]` = *expansion-initializer*; where *n* is the number of elements required in a valid *identifier-list* for such a structured binding declaration, and is equivalent to:

```
{
    init-statement
    constexpr-specifier_opt auto&& seq = expansion-initializer ;
    {    //  iᵗʰ repetition of the substatement
        for-range-declaration = get-exprᵢ ;
        statement
    }
}
```

where *get-expr$_i$* is the *initializer* for the ith *identifier* in the corresponding structured binding declaration. The *constexpr-specifier* is present in the declaration of `seq` if `constexpr` appears in *for-range-declaration*. The name `seq` is used for exposition only.

10. [*Example 1*:

```
struct S { int i; short s; };
consteval long f(S s) {
    long result = 0;
    template for (x: s) {
        result += x;
    }
    return result;
}
static_assert(f(S{1, 2}) == 3);
```

— *end example*]

11. [*Example 2*:
```
consteval int f(auto... Containers) {
  int result = 0;
  template for (auto c: { Containers... })
    result += c[0];
  return result;
}
constexpr int c1[] = {1, 2, 3};
constexpr int c2[] = {4, 3, 2, 1};
static_assert(f(c1, c2) == 5);
```
— *end example*]

12. [*Example 3*:
```
template <typename T> consteval std::optional<int> f() {
  constexpr std::vector statics = static_data_members_of(^T);
  template for (constexpr std::meta::info s : statics)
    if (name_of(s) == "ClsId")
      return [:s:];
  return std::nullopt;
}
struct Cls { static constexpr int ClsId = 14; };
static_assert(f<Cls>() == 14);
```
— *end example*]

8.78.2 The break statement [stmt.break]

Modify paragraph 1 as follows:

1. A `break` statement shall be enclosed by (_stmt.pre_) an *iteration-statement* (_stmt.iter_) or a `switch` statement (_stmt.switch_). The `break` statement causes termination of the smallest such enclosing statement; control passes to the statement following the terminated statement, if any. A `break` statement enclosed by an *expansion-statement* for which the smallest enclosing *iteration-statement* or `switch` statement is not a substatement of that *expansion-statement* is ill-formed.

8.78.3 The continue statement [stmt.cont]

Modify paragraph 1 as follows:

1. The `continue` statement shall be enclosed by (_stmt.pre_) an *iteration-statement* (_stmt.iter_). The `continue` statement causes control to pass to the loop-continuation portion of the smallest such enclosing statement, …

A `continue` statement enclosed by an *expansion-statement* for which the smallest enclosing *iteration-statement* is not a substatement of that *expansion-statement* is ill-formed.

9.6 Structured binding declarations [dcl.struct.bind]

Add the following paragraph:

6. An expression `E` is *destructurable* if the declaration
    `auto [identifier-list] = E;`
is valid for some *identifier-list*.

9.12.6 Fallthrough attribute [dcl.attr.fallthrough]

Add the following:

1. The attribute-token fallthrough may be applied to a null statement; such a statement is a fallthrough statement. No attribute-argument-clause shall be present. A fallthrough statement may only appear within an enclosing switch statement ([stmt.switch]). The next statement that would be executed after a fallthrough statement shall be a labeled statement whose label is a case label or default label for the same switch statement and, if the fallthrough statement is contained in an iteration statement or expansion statement, the next statement shall be part of the same execution of the substatement of the innermost enclosing iteration statement or expansion statement. The program is ill-formed if there is no such statement.

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

Add the following (and renumber accordingly):
    …
(3.8) — a *conversion-function-id* that specifies a dependent type, or
(3.9) — it is a name introduced by a *for-range-declaration* that contains a placeholder type and is declared in an *expansion-statement* (_stmt.expand_) for which the *expansion-initializer* is not an iterable (_stmt.ranged_) expression, or
    …

13.8.3.4 Value-dependent expressions [temp.dep.constexpr]

Add the following (and renumber accordingly):
    …

(2.3) — it is the name of a non-type template parameter,

(2.4) — it is a name introduced by a *for-range-declaration* declared in an *expansion-statement* for which the *expansion-initializer* is an iterable expression,

...

15.10 Predefined macro names [cpp.predefined]

Add the following entry to Table 17:

...	...
`__cpp_expansion_statements`	202406L
...	...