Core Issue 3123

"Global lookup for `begin` and `end` for expansion statements"

Document #: P4026R0
Date: 2026-02-23
Audience: EWG
Author: Vlad Serebrennikov
<serebrennikov.vladislav@gmail.com>

Wording of expansion statements

[stmt.expand] paragraph 4:

An expansion statement is

— an *enumerating expansion statement* if its *expansion-initializer* is of the form *expansion-init-list*;

— otherwise, an *iterating expansion statement* if its *expansion-initializer* is an expansion-iterable expression;

— otherwise, a *destructuring expansion statement*.

Status quo of the Working Draft

[stmt.expand] paragraph 3:

An expression is *expansion-iterable* if <...> argument-dependent lookups for `begin(E)` and for `end(E)` each find at least one function or function template.

```
namespace N {
  struct S1 { int array[8]; };
  template <typename T> int* begin(T&);
  int* end(S1&);

  struct TupleLike {
    int first;
    int second;
  };
}
void f1() {
  // error: no viable candidate for end()
  template for (auto i : N::TupleLike{});
}
```

Proposed resolution

[stmt.expand] paragraph 3:

An expression is *expansion-iterable* if <...> argument-dependent lookups for `begin(E)` and for `end(E)` each find at least one ~~function or function template~~ viable candidate.

```
namespace N {
  struct S1 { int array[8]; };
  template <typename T> int* begin(T&);
  int* end(S1&);

  struct TupleLike {
    int first;
    int second;
  };
}
void f2() {
  // OK, destructuring
  template for (auto i : N::TupleLike{});
}
```

Alternative approach

[stmt.expand] paragraph 3:

An expression is *expansion-iterable* if <...> argument-dependent lookups for `begin(E)` and for `end(E)` each find at least one ~~function or function template~~ viable candidate, and overload resolution succeeds for both.

```
namespace M {
  struct B {};
  template <typename T> int* begin(T&);
}
namespace N {
  struct S1 : M::B { int array[8]; };
  template <typename T> int* begin(T&);
  int* end(S1&);
}
void f3() {
  // error: ambiguous call to 'begin'
            for (auto i : N::S1{});
  // OK, destructuring
  template for (auto i : N::S1{});
}
```

Potential consequences 1: `members_of`

Specializations are invisible for `members_of` ([meta.reflection.member.queries]/2.4).

```
struct S1 { int array[8]; };
template <typename T> int* begin(T&);
int* end(S1&);

struct TupleLike {
  int first;
  int second;
};

constexpr auto ctx = std::meta::access_context::current();
constexpr int count1 = members_of(^^::, ctx).size();
void f4() {
  template for (auto i : N::TupleLike{}); // OK, destructuring
}
constexpr int count2 = members_of(^^::, ctx).size();
static_assert(count1 == count2); // OK in both Clang fork and GCC
```

Potential consequences 2: `define_aggregate`

Constant expressions that are evaluated during template instantiation are separate from `consteval` block that encloses expansion statement, and can't rely on it ([temp.arg.nontype]/2, [expr.const]/9.30).

```
struct S1 { int array[8]; };
struct S2;

template <typename T>
constexpr
std::enable_if_t<size_of(define_aggregate(^^S2, { // error
  data_member_spec(^^int, {.bit_width=sizeof(T)})
})), int*>
begin(T&);

int* end(S1&);

struct TupleLike {};
consteval {
  template for (auto i : TupleLike{});
}
```

Potential consequences 3: non-SFINAEble errors

Well-formed destructuring expansion statements will be ill-formed because of errors outside of immediate context while checking for iteration expansions statement. Same is true for range-based `for`, though.

```
namespace N {
  struct S1 { int array[8]; };

  template <typename T>
  requires([] {
    static_assert(false, "range-based for was used!");
    return false;
  }());
  int* begin(T& s);

  struct TupleLike {};
}
void f6() {
  for (auto i : N::S1{}); // error
  template for (auto i : N::TupleLike{}); // error
}
```

Summary

- Status quo is problematic and is worth fixing
- Core-level solutions involve additional template instantiations
  - Additional instantiations and semantic analysis are never free
  - Side effects of instantiations should not pose problems
  - But they are vulnerable to friend injection (like everything else)
- Proposed resolution is better than the alternative
  - No difference in most cases, but it follows range-based `for` in corner cases