Document: **P1306R1**
Revises: **P1306R0**
Date: **01-21-2019**
Audience: **EWG**
Authors: **Andrew Sutton (asutton@uakron.edu)**
Sam Goodrick (sgoodrick@lock3software.com)
Daveed Vandevoorde (daveed@edg.com)

Expansion statements

Version history

r1 This paper unifies the different form of expansion statements, so that only one syntax is needed. We have further refined the semantics to ensure that expansion can be supported for all traversable sequences, including ranges of input iterators. We also added discussion about break and continue within expansions.

r0 The original version of this paper is P0589r0, Tuple-based for loops. We have modified the original proposal to work with more destructurable objects including classes and parameter packs. We have also added a constexpr-for version 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 (see P1240r0).

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, parameter pack, or range. Any facility that needs to traverse the elements of a heterogeneous container inevitably duplicates this kind of repetition using recursively instantiated templates, which allows some part of the repeated statement to vary (e.g., by type or constant) in each instantiation.

While this behavior can be encapsulated in a single operation (e.g., Boost.Hana's `for_each` template), there are a number of reasons we would prefer language support. First, repetition is a fundamental building block of algorithms. We should be able to express that concept directly rather than through recursively instantiated templates.. Second, we'd like that repetition to be as inexpensive as possible. Recursively instantiating templates generates a large number of template specializations, which can end up consuming a lot of compiler memory and compile time. Finally, we'd like the ability to "iterate" over both destructible classes and parameter packs, and both effectively require language support to implement correctly.

Basic usage

Here is an example of iterating over the elements of a tuple using the Hana library:

```
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 print each element of the tuple in turn, by calling the generic lambda. Each call instantiates a new function containing a call to `cout` for the corresponding tuple element.

Using the feature described in this proposal, that code could be written like this:

```
auto tup = std::make_tuple(0, ‘a’, 3.14);
for... (auto elem : tup)
  std::cout << elem << std::endl;
```

The 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 this:

```
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;
}
```

In other words, an expansion statement *is not a loop*. It is a repeated version of the loop body, in which the loop variable is initialized to each successive element in the tuple. Because the loop variable is re-declared in each version of the loop 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:

- Tuples (as above)
- Arrays
- Destructurable classes
- Unexpanded argument packs
- Constexpr ranges (described below)

Note that it is also possible to define expansion over a *brace-init-list*, but we have opted not to provide that functionality at this time.

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).

```
template<HashAlgorithm H, StandardLayoutType T>
bool hash_append(H& algo, const T& t) {
  constexpr meta::info members = meta::data_members_of(reflexpr(T));
  for... (constexpr meta::info member : members)
    hash_append(h, t.idexpr(member));
}
```

Here, `constexpr` appears as *decl-specifier* of the loop variable `member`, meaning that in each expansion, that value is a constant expression (i.e., suitable for use in a template argument list). This is necessary since we that variable with the `idexpr` operator, which yields a resolved reference to the corresponding data member. (This is similar to requesting a pointer-to-member, except that `idexpr` also works with bit fields.)

Note that `data_members_of` returns a `constexpr` range: a forward-traversable sequence of `meta::info` values that describe the data members of `T` (or rather whatever type `T` becomes when the template is instantiated. The fully expanded statement is roughly equivalent to this:

```
{
  constexpr member0 = *std::next(std::begin(members), 0);
  hash_append(h, t.idexpr(member0));
}
{
  constexpr member1 = *std::next(std::begin(members), 1);
  hash_append(h, t.idexpr(member0));
}
...
{
  constexpr memberK = *std::next(std::begin(members), K);
  hash_append(h, t.idexpr(member0));
}
```

The expansion terminates after *K* expansions, where *K* is `std::distance(std::begin(), std::end())`.

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

Without the ability to use an expansion statement, we need a recursive function template that traverses a list of reflections. That implementation, based on an earlier version of the forthcoming static reflection proposal is shown below:

```
// Recursive template
template<HashAlgorithm H, StandardLayoutType T, meta::info X>
  requires meta::is_class(X)
hash_append_impl(H& h, T const& t) {
  // Visit the current member (hash it if you can).
  if constexpr(!meta::is_invalid(X)) {
    if constexpr (meta::is_non_static_data_member(X))
      hash_append(h, t.idexpr(X));
  }
  // Continue hashing until we run out of members.
  if constexpr(!meta::is_invalid(meta::next(X)))
    hash_append_impl<H, T, meta::next(X)>(h, t);
}

// Main interface
template<typename H, typename T>
std::enable_if_t<std::is_class<T>::value, void>
hash_append(H& h, T const& t) {
  hash_append_impl<H, T, meta::front(reflexpr(T))>(h, t);
}
```

In this implementation `meta::front` and `meta::next` are used to iterate (statically) over the members of a declaration. They are not included in our current static reflection proposal since they are no longer needed.

Break and continue

At this time, we are proposing to disallow `break` and `continue` within expansion statements. These can be re-added as needed. Their meaning is easy to define and implement. Our main concern is that users will confuse these statements as providing some kind of control over the expansion itself (they would not).

Syntax and semantics

The syntax for an expansion statement is similar to that of a range-based for loop, except that `...` follows the `for` keyword.

```
for ... (for-range-declaration : expansion-initializer) statement
```

An *expansion-statement* expands statically to a statement that is equivalent to the following pattern.

```
{
  constexpr-specifier range-initializer-declaration __range = expansion-initializer;
  constexpr-specifier auto __begin = begin-expr;
  constexpr-specifier 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
  }

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

The placeholder *constexpr-specifier* is the token `constexpr` if the *for-range-declaration* includes `constexpr` in its *decl-specifier-seq*; otherwise, it is replaced by a space (i.e., not present). The *range-initializer-declaration* is `auto&&` if the *expansion-initializer* has array or function type, and `auto` otherwise (this prevents decay for prvalues of those types). The meaning of placeholder expressions *begin-expr*, *end-expr*, *get-expr*, *next-expr* depend on the type of the *expansion-initializer* and the presence of the `constexpr` keyword in the loop head.

If *expansion-initializer* contains unexpanded parameter packs, repetition is defined over an index `I` into the argument pack, and the named expressions are:

- *begin-expr* is `0u`
- *end-expr* is `sizeof...(`*expansion-initializer*`)`
- *get-expr(*`I`*)* is the `I`th argument (expression) in the pack.
- *next-expr(*`I`*)* is `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` ranged over by *expansion-initializer*, and the placeholder expressions are:

- *begin-expr* and *end-expr* are that of the range-based for loop,
- *get-expr(*`I`*)* is `*I`
- *next-expr(*`I`*)* is `std::next(I)`

Lastly, 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 `I`th entity named by the structured binding
- *next-expr(*`I`*)* is `I + 1`

Note that the form of the expansion is intended to be valid for any expandable entity used with the loop. In the most general case, this emulates the hand-unrolling of 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 call `std::distance` or advancing multiple elements in a single call to `std::next`. For unexpanded packs, and destructurable objects, 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 they can determine the iterator category of the range.

Examples:

```
auto tup = std::make_tuple(0, 'a');
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;
    }
  }
```

Here, the iterators have been elided since the “iterator” can be maintained internally by the implementation.

```
  template<typename... Ts>
  void f(Ts&&... args) {
    for... (const auto& x : args)
      cout << x << ‘\n’;
  }

  void foo() {
    f(0, ‘a’);
  }
```

The instantiation of `f` generated from `foo` will have the expansion:

```
  {
    {
      const auto& x = /* first element args */;
      cout << x << ‘\n’;
    }
    {
      const auto& x = /* second element in args */;
      cout << x << ‘\n’;
    }
  }
```

Below is an example of a `constexpr` expansion:

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

… and its expansion:

```
  {
    constexpr auto __range&& = vec;
    constexpr auto __end = vec.end();

    constexpr auto __iter_0 = vec.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, potential additions, and implementation notes.

Required header files

This feature does not require users to include additional header files to use the expansion facilities, just like the range-based for loop. Many expansions are defined in terms of 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*.

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:

```
  for... (auto x : enumerate(some_tuple)) {
    // x has a count and a value
    std::cout << x.count << ": " << x.value << std::endl;

    // 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. This facility should be relatively easy to implement.

Interaction with initializer lists and parameter packs

The feature could be extended to allow *brace-init-list*s in the *expansion-initializer*. This is currently ill-formed since it requires deduction from an initializer list. However, there may be some value in supporting this syntax:

```
  for... (auto x : {0, ‘a’, 3.14})
    std::cout << x;
```

which would be equivalent to:

```
  for... (auto x : make_tuple(0, ‘a’, 3.14))
    std::cout << x;
```

We are not formally proposing these extensions at this time since they would (could?) potentially introduce a new form of template argument deduction in order to avoid an explicit rewrite to `make_tuple`.

Implementation experience

At the time of writing, the foundations of the feature have been implemented in a fork of Clang 8.0.0, except for the unified syntax (constexpr expansions still require the constexpr keyword instead of the...). However, both statements use the same underlying implementation to choose salient operations for expansion. The SSA-style expansion for input ranges is also unimplemented as it requires a non-trivial change to our approach.

For these *expansion-statements* to work, the body of the loop must be parsed as if inside a template and then repeatedly instantiated after the body is parsed. Moreover, names appearing in expressions within an expansion loop body may not be ODR-used, even in a non-dependent context. If the expansion operand is empty, the result of expansion is an empty statement: the statements, expressions, and declarations within the body 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).