Document: P1306R0
Revises: P0589R0
Date: 08-10-2018
Audience: EWG
Authors: Andrew Sutton (asutton@uakron.edu)
Sam Goodrick (sgoodrick@lock3software.com)
Daveed Vandevoorde (daveed@edg.com)

Expansion statements

Version history

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 << member << std::endl;
```

The output of the program is the same. However, the familiar syntax makes the algorithm easier to understand and easier to write.

Repeated 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 namespace meta = std::meta;
template<HashAlgorithm H, StandardLayoutType T>
  bool hash_append(H& algo, const T& t) {
    constexpr auto data_members =
        members_of(reflexpr(T), meta::is_nonstatic_data_member);
    for constexpr (meta::info member : data_members)
      hash_append(h, t.unreflexpr(member));
  }
```

Note the use of constexpr for in this algorithm. We need member to be a constant expression so that it works with the `unreflexpr` operator. Used here, that operator resolves to the corresponding subobject (or bitfield!) of `t`.

We could not have used for... in this application for two reasons. First, basic expansions do not work ranges (it is not generally possible to infer compile-time bounds from a range). Second, the loop variable `member` would not be a constant expression, so we couldn't use it with unreflexpr or any reflection facility, for that matter.

Without the ability to use an expansion loop, 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.unreflexpr(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.

Syntax and semantics

There are two forms of expansion-statements.

```
for ... (element-declaration : expansion-initializer) statement
for constexpr (element-declaration : expansion-initializer) statement
```

The first allows the repetition of a statement for members of tuples, arrays, some classes, and parameter packs. The second allows repetition where the “loop variable” is a constant expression, which also allows ranges to be used within the loop.

The break and continue statements are not valid within an expansion-statement.

The *expansion-initializer* is an expression with one of the following properties:

- it contains unexpanded parameter packs, or
- it can appear as an initializer of a structured binding declaration, or if not that,
- it is a *constant expression* that can be used as the *range-for-initializer* in a range-based for loop within a constexpr function, in which case the statement must begin with `for constexpr`.

In each case, the *expansion-initializer* denotes a compile-time sequence of *N* elements for which the body can be repeated, once for each element. We propose two distinct forms because some expressions satisfy both properties (e.g., arrays are both destructurable and iterable). The rules used to define the expansion are ordered so there is no ambiguity.

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

```
{
  constexpr-spec auto &&__range = expansion-initializer;
  constexpr-spec auto __end = end-expr;

  constexpr auto __iter_0 = begin-expr;
  <stop expansion if __iter == __end>
  {
    constexpr-spec element-declaration = get-expr(__iter_0)>;
    statement
  }

  constexpr auto __iter_1 = next-expr(__iter_0);
  <stop expansion if __iter == __end>
  {
    constexpr-spec element-declaration = get-expr(__iter_1)>;
    statement
  }

  // ... repeats N - 2 times
}
```

The placeholder *constexpr-spec* is the token `constexpr` if the statement is spelled `for constexpr`. Otherwise, it is replaced by a space (i.e., not present). The meaning of placeholder expressions *begin-expr*, *end-expr*, *get-expr*, *next-expr* depend on the properties 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 abstract (conceptual) index `I` into the sequence of arguments in the pack.

- *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 *expansion-initializer* has type "array of `N T`", repetition is defined over a compile-time integer index.

- *begin-expr* is `0u`
- *end-expr* is `N`
- *get-expr(*`I`*)* is `__range[I]`
- *next-expr(*`I`*)* is `I + 1`

Otherwise, if the type of *expansion-initializer* satisfies the requirements of `Tuple` (see below), repetition is defined over a compile-time integer index.

- *begin-expr* is `0u`
- *end-expr* is `std::tuple_size_v<decltype(__range)>`
- *get-expr(*`I`*)* is `std::get<I>(__range)`
- *next-expr(*`I`*)* is `I + 1` where I is the current tuple index

Otherwise, *expansion-initializer* has class type satisfying the requirements for destructuring, and repetition is defined over an abstract (conceptual) index `I` into the sequence of *N* non-static data members in the complete object.

- *begin-expr* is `0u`
- *end-expr* is *N*
- *get-expr(*`I`*)* is the expression $\texttt{__range.m}_{\texttt{I}}$, where $\texttt{m}_{\texttt{I}}$ is the path to the `I`th direct subobject
- *next-expr(*`I`*)* is `I + 1`

Otherwise, if the `constexpr` keyword is present in the the loop head and expansion-initializer satisfies the requirements of `ConstexprRange` (see below), the repetition is defined over the range's iterator.

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

For the purpose of this proposal a `Tuple` type `T` has the following valid expressions and associated types, with `t` being an object of type `T`:

- `std::tuple_size_v<T>` is a valid expression
- `std::get<N>(t)` is a valid expression when `0` <= `N` < `std::tuple_size_v<T>`

(A more complete concept for `Tuple`s could be defined; only the operations needed for expansion are listed here.)

A `ConstexprRange` is `Range` whose begin and end operations are `constexpr` and whose iterator type is a literal type with all salient iterator operations being `constexpr`.

Note that the declarations of the range and iterators are provided for exposition only. For example, it doesn't make sense to declare a range variable for an initializer list, and any repetition based on an integer counter can be maintained by the implementation (i.e., for packs, arrays, tuples, and classes). However, all of these declarations *are* needed for range-based expansion.

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;
```

And similarly for constexpr expansion:

```
for constexpr (auto x : {0, ‘a’, 3.14})
  do_something<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. Expansion statements of `for...` variety as well as their extension to support arrays and some classes have been implemented.

The implementation of `std::tuple` expansion statements has been completed and expands to example 1 in section *Syntax and semantics*. Array expansion statements work nearly identically to `std::tuple` expansion statements, except that the element-wise expansion would use an array-index expression, rather than a call to `std::get`. Expanding over a destructured class is a work in progress. We have not yet implemented `constexpr for`.

For these expansion-statements to work, the body of the loop must be parsed as if inside a template and then repeatedly instantiated. The reason for this is that the body (usually) depends, in some way, on the types or constant values associated with each element in the *expansion-initializer*. This is true even for ranges in a `for constexpr` statement, where all elements have the same type. Because the loop variable is a constant expression, it can be used to introduce dependent types indirectly. The static reflection example earlier in this paper includes such an example.