Doc. No: P0589R0
Date: 2017-02-04
Group: EWG
Contact: Andrew Sutton <asutton@uakron.edu>

Tuple-based for loops

Introduction

This paper proposes to enable the iteration of tuple-members using the syntax of the range-based for loop. The goal is to make it easier for programmers to write algorithms for heterogeneous containers.

Current practice requires the use of a for-each algorithm and a generic function object or generic lambda to define behaviors. Here is a simple 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. Using 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 much easier to understand and easier to write. Furthermore, it avoids the need for lambda capture, allowing local variables to be used directly.

One of the major motivations for proposing this extension is related to the static reflection proposal P0590r0. Reflecting e.g., the members of a class yields a tuple-like object that can then be used as the range of a for loop. For example, we could print the member names of some class S like this:

```
for (auto member : $S.members())
  std::cout << member.name() << std::endl;
```

The meaning of the algorithm should be obvious.

Semantics

Tuple-based for loops are not actually loops; the body of the loop is instantiated once for each element in the tuple. Consider the following loop.

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

This is equivalent to:

```
{
  auto&& __tuple = tup;
  {
    auto elem = std::get<0>(__tuple);
    std::cout << member << std::endl;
  }
  {
    auto elem = std::get<1>(__tuple);
    std::cout << member << std::endl;
  }
  {
    auto elem = std::get<2>(__tuple);
    std::cout << member << std::endl;
  }
}
```

The semantics of the extension are relatively straightforward. The meaning of the `for` loop depends on the type of the *for-range-initializer* T. If T is non-dependent, then:

- If T satisfies the `Range` concept, the loop is a range-based `for` loop.
- If T satisfies the `Tuple` concept, then the loop is a tuple-based `for` loop.
- Otherwise, the program is ill-formed.

By "satisfies the concept", I mean that lookup is used to construct certain valid expressions. The `Range` concept is satisfied when the *begin-expr* and *end-expr* can be formed using the existing rules for the range-based `for` loop.

The `Tuple` concept is satisfied when T has class type and the following invented declaration is well-formed:

```
constexpr std::size_t N = std::tuple_size<T>::value
```

Note that the concept does not check for a `get` function since that requires a concrete template argument and even 0 may cause the program to be ill-formed (i.e., when `N == 0`). Some tuple-like implementations may disable out-of-bounds `get` functions for overload resolution rather than statically asserting the condition.

When T is a Tuple, the loop range-based `for` statement is initially equivalent to:

```
{
  auto&& __tuple = for-range-initializer;
  loop-body
}
```

where *loop-body* is the *compound-statement*:

```
{
  for-range-declaration = get-expr;
  statement
}
```

The *loop-body* is parameterized by an invented non-type template parameter with type `std::size_t`. The expression *get-expr* is `get<I>(__tuple)` where `get` is looked up in the associated namespace of `__tuple`. Ordinary unqualified lookup is not performed. If the lookup of `get` yields no candidates, the program is ill-formed.

The *loop-body* is instantiated for each integer value K in the range [0, N) by substituting K for `I`. If any substitution in the *loop-body* fails, the program is ill-formed. The range-based `for` statement is finally equivalent to the sequence of instantiated *loop-body*s.

The `break` and `continue` statements have slightly different meaning within a tuple-based for loop. The `break` statement passes control to the statement following the last instantiated *loop-body*, if any. The continue statement passes control to the next instantiated *loop-body*, if any. For example, this loop

```
for (int x : tup) {
  if (x == 0) continue;
  if (x == 1) break;
}
```

is equivalent to this sequence of statements:

```
{
  auto&& __tuple = ...;
  {
    __loop_0:
    if (x == 0) goto loop_1;
    if (x == 1) goto loop_end;
  }
  {
    __loop_1:
    if (x == 0) goto loop_2;
    if (x == 1) goto loop_end;
  }
  // ...
  {
    __loop_N:
    if (x == 0) goto loop_end;
    if (x == 1) goto loop_end;
  }
  loop_end:
}
```

Observations and notes

Preserves the meaning of existing code

This proposed feature does not change the meaning of existing code. Range-based `for` loops continue to be range-based (i.e., not tuple-based) because the `Range` concept check is given precedence in the semantics of the loop.

Extra header files

This feature does not require users to include headers—sort of. If a programmer wants to iterate over a `std::tuple`, then they will have already included the header in order to construct the tuple object.

Furthermore, defining a model of the `Tuple` concept requires a partial specialization of `std::tuple_size`, so all implementations would either have included the `<tuple>` header already or provide a forward declaration. As before, iterating over a tuple-like object would require the user to have previously include the appropriate header.

Lookup on `std::tuple_size` is only performed when the range type is non-dependent, which means that generic algorithms can use the syntax without including any additional headers.

Unrolling array loops

Range-based `for` loops over arrays continue to iterate in the usual way. However, this facility can be used to explicitly unroll loops. If an array could be "converted" to a tuple, loops over that container would instantiate the body once for element. This can be done using an `unroll` facility.

```
int a[] { 0, 1, 2, 3 };
for (int& n : unroll(a))
  n *= 2;
```

This loop is equivalent to:

```
int a[] { 0, 1, 2 };
{
  auto&& __tuple = unroll(a);
  {
    int& n = get<0>(__tuple);
    n *= 2;
  }
  {
    int& n = get<1>(__tuple);
    n *= 2;
  }
  {
    int& n = get<2>(__tuple);
    n *= 2;
  }
}
```

Although most compilers would probably be able to unroll such a simple loop automatically, it may not be possible with more complex control structures.

The unroll function can be defined like this:

```
template<typename T, int N>
auto unroll(T(&arr)[N]) {
  return homogenous_tuple<T, N>(arr);
```

```
   }
```

The homogeneous_tuple class is essentially std::array, but satisfying the Tuple concept and not the Range concept. This could also be extended to work for any Range with compile-time size.

A similar technique could be used to unroll compile-time integer sequences.

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 value
  std::cout << x.count << ": " << x.value << '\n';

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

The enumerate function returns a simple tuple adaptor whose elements are count/value pairs. This facility should be relatively easy to implement.

Interaction with concepts

Concepts can be used in the declaration of the loop variable to provide deduction guarantees for tuple elements:

```
for (Number& n : some_tuple)
   n *= 2;
```

If deduction of the loop variable fails during instantiation, the program would be ill-formed, presumably with a reasonably good-looking error message.

Interaction with constexpr-if

Loop bodies can include conditionally compiled branches further simplifying algorithms creation for heterogeneous containers. This would work particularly well with concepts.

```
for (auto& x : some_tuple) {
  using T = decltype(x);
  if constexpr (Number<T>) // do number stuff
  if constexpr (String<T>) // do string stuff
}
```

This can also be used with return type deduction to define algorithms whose result types depend on one or more elements of the tuple.

The tuple-based for loop does not directly support the computation of types based on the values of elements in a tuple. Algorithms that aim to compute projections or transformations on the types and values of heterogeneous containers (i.e., tuples) cannot be implemented using a tuple-based for loop and constexpr if.

It should be possible to use the tuple-based for loop to compute types, except that C++ does not provide direct support for type variables. Such a feature could be used to compute

```
typename R = tuple<>
for (auto& x : some_tuple) {
  using T = decltype(x);
  if constexpr (is_integral_v<T>)
    R = append_t<R, T>;
}
```

After instantiation, the type R would be a tuple comprised of the integral types of `some_tuple`.

Type variables are way beyond the scope of this proposal (but potentially a very interesting direction to explore).

Interaction with initializer lists and parameter packs

The feature could be extended to allow more *brace-init-list*s in the *for-range-initializer*. Currently, the elements of such a list are required to have the same type because the deduction produces a `std::initializer_list`.

It might be worthwhile to define the semantics of a for-loop over a *brace-init-list* to use a tuple-based expansion. That is, this loop

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

would be equivalent to

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

This change to the semantics would also allow this:

```
template<typename... Args>
void f(const Args&... args) {
  for (auto x : {args...})
    std::cout << x;
}
```

The function parameter pack would expand within the *brace-init-list* and the loop would be instantiated once for each element. Alternatively, we may consider supporting expansions directly in the *for-range-initializer*.

```
template<typename... Args>
void f(const Args&... args) {
  for (auto x : args...)
    std::cout << x;
}
```

This should be equivalent to the previous version of the function.

Implementation experience

Yes. Work is ongoing. At the time of writing, the foundations of the feature have been implemented (tuple lookup and loop body instantiation).

Acknowledgements

Thanks to Louis Dionne for comments on an early draft and pointing that I had completely overlooked the meaning of `break` and `continue` statements in tuple-based `for` loops.