multi-range-based for loops

Document Number: P0026
Date: 2015-08-11
Project: Programming Language C++, Evolution Working Group
Reply-to: Matthew McAtamney-Greenwood (izzy.coding@outlook.com)

Introduction

The new range-based for loops have become a very useful tool since being introduced in C++11. This proposal is to extend its use with the ability for allowing 2 or more ranges to be part of the loop iteration mechanism.

Motivation and Scope

Currently there is no simple way to iterate over multiple containers simultaneously and perform a given task on each ranges relative iteration values.

For example:

If I have 2 vectors of data and I need to iterate over them and pass each set of values to a third party function, I currently have the following options:

1) Using a while loop

```
auto iter1 = vec1.begin();
auto iter2 = vec2.begin();
while (iter1 != vec1.end() && iter2 != vec2.end())
{
    DoSomething(*iter1, *iter2);
    ++iter1;
    ++iter2;
}
```

2) Using legacy for loop

```
for (auto iter1 = vec1.begin(), iter2 = vec2.begin();
        iter1 != vec1.end() && iter2 != vec2.end();
        ++iter1, ++iter2)
{
    DoSomething(*iter1, *iter2);
}
```

3) Using current range-based for loop with boost::zip_iterator

```
template <typename Cont...>
auto multi_range(Cont&... containers) ->
decltype( boost::make_iterator_range(
        boost::make_zip_iterator(
            boost::make_tuple(containers.begin()...)
        ),
        boost::make_zip_iterator(
            boost::make_tuple(containers.end()...)
        ) ))
{
    return {
        boost::make_zip_iterator(
            boost::make_tuple(containers.begin()...)
        ),
        boost::make_zip_iterator(
            boost::make_tuple(containers.end()...)
        ) };
}
```

```
for (auto&& t : multi_range(vec1, vec2))
{
    DoSomething(t.get<0>(), t.get<1>());
}
```

As you can see, although these options do what is required, none of them are easy to understand at a glance. The best solution is currently option 3, which given the boilerplate template function does provide a simple for statement implementation.

However, that approach is obscured by the fact the iteration variables do not describe what the underlying values are, as they need extracting from the internal tuple explicitly. This means there is a requirement to know more about the context than that which can be expressed directly in the code.

Also with all the verbosity it is very easy to introduce bugs due to typos. Not only could these buggy loops compile and run with potentially undefined behaviour, but they could also introduce potentially hidden implicit conversions that could also trigger further undefined behaviour.

I am also aware of other proposal discussions that would allow for multiple return value unpacking to a named tuple-like structure. This would improve the boost::zip_iterator version, however it still leaves a lot of potential for undefined behaviour depending on the implementation of the multi_range. Having said that I do believe that work could go hand-in-hand with this proposal in a future revision in order to provide the unpacking ability for complex groups of ranges similar to that of the multi_range type function defined above, or with the work being done on compile-time reflection, could be potentially extended to unpack values for any library/user-defined types. However this is currently out of the scope of this proposal. I merely wanted to note it here as I have taken these possibilities into consideration for this proposals design.

Another possible alternative to this proposal would be to have library algorithms with multiple range support. This would allow code similar to:

```
for_each( range_a, range_b, range_c, []( A&& a, B&& b, C&& c ) { ... } );
```

Having this library feature however I think would not be easy to implement given the current state of the standard. The easiest possible implementation would require syntax similar to the following:

```
for_each( function, ranges... );
```

or

```
for_each( multi_range { ranges... }, []( A&& a, B&& b, C&& c) { ... } );
```

The first however would not follow the current format of the existing library for_each algorithm, and the second would require more knowledge from the developer due to the use of a wrapping multi-_range type. To implement this in the correct order would require special processing to unpack the parameters (an example of which can be found in the related links [6]).

```
for_each ( range-and-function-args... );
```

Also as a library feature this would most likely take iterators and thus would be more verbose. However if the library version could take ranges it would be difficult to implement the special begin/

end lookup rules for range-for (in particular, the final step performs an ADL-only lookup that is not used anywhere else in C++), introducing subtle semantic differences.

Another possibility would be to implement something like the following:

```
template<Callable Fn, typename T...>
void for_each(T&& t..., Fn&& fn)
{
    auto ranges = make_autorange( forward(t)... );
    for( ; ranges.current() != ranges.end(); ranges.next() )
    {
        fn(*ranges.current()...);
    }
}

template<Iterator B, Iterator E>
auto make_autorange(B&& begin, E&& end)
{
    return make_autorange( range{ forward(begin), forward(end) } );
}

template<Range R...>
auto make_autorange(R&& range...)
{
    return autorange{ forward(range)... };
}
```

Where `make_autorange` could be specialised to implement current existing for_each range variants. This however requires a dependency on the concepts TS/proposal(s), as well as a concept definition for iterators and also the creation of a new tuple-like type providing `current()` which exposes a collection of the current iterators, `end()` which would expose a collection of ending iterators, whereby the comparison of both collections would return true if any related current iterator equals its respective end iterator. Also the `current()` function would require the ability to unpack its resultant collection in order to pass all required parameters into the Fn callable function object (usually a lambda function).

Another advantage to using this proposal over the library feature implementation is the ability to use `break` and `return` statements within the loop body to terminate early, where the library version would require the abuse of exceptions. Also this language feature I believe would be easier to teach, read, write and understand as well as being a natural extension of existing syntax that does not require the use of function pointers or lambda expressions. Also there would be no library dependancies so this feature would be usable on freestanding implementations that do not have <algorithm> or <iterator> headers available. Having said that I feel that this would be potentially useful to have in addition to the language feature and should be proposed separately.

I am sure there are other possible uses for this ability within the standard library, however, I don't have the knowledge to identify these. Therefore I leave this open for discussion. Also I believe this ability might also allow for new optimisation abilities, although I don't have enough knowledge to speculate on these either.

Impact on the Standard

This proposal is a core C++ extension. It does not require changes to any standard classes, functions or headers. It does however require changes in the definition of the C++11 range-based for loop specification.

I have not fully checked, however, there may be some library use cases for this proposal.

Design Decisions

This feature is designed to be as close to the existing interface as possible and only introduces a semi-colon operator to allow for multiple ranges to be specified along with their declared variable in order to name each ranges respective value within each iteration. The syntax is taken from the initial request by Gabriel Dos Reis [EWG 43]

```
for (auto&& val1 : cont1; auto&& val2 : cont2)
{
    DoSomething(val1, val2);
}
```

This multiple form essentially changes only the iteration process of the for loop. Each iteration will increment an iterator into each of the ranges specified and their subsequent declared variable should be initialised to the result of dereferencing the iterator for use within the for statement body.

In the case where mismatching container sizes are passed to the for statement, the end of iteration should at the point of hitting the first of any range ends. This would then eliminate any invalid range iterators.

This syntax is much clearer and is easier to reason about than the existing current options. This syntax would also eliminate the possibility for typos in the for statement declaration and body where potential invalid iterator comparisons and implicit type conversions could change the behaviour of the code but still compile without warnings or errors.

Although I have no knowledge of compilers, I would also assume there would be possibilities for extra optimisations at compile-time.

The specification below also incorporates a small change proposed in N4382 that allows begin and end iterators of different types. However, this it's not essential to the proposal and trivial to remove if desired.

This proposal however should not limit the possible implementation of the multiple return value discussion proposals (if/when one is proposed). I have left the possibility open for this based on current discussions. However that is currently out-of-scope for this current proposal.

Technical Specifications

Change 6.5 paragraph 1 to:

```
Iteration statements specify looping.
    Iteration-statement:
        while ( condition ) statement
        do statement while ( expression ) ;
        for ( for-init-statement condition(opt); expression(opt))
statement
        for ( for-range-statement ) statement
    for-init-statement:
        expression-statement
        simple-declaration
    for-range-statement:
        for-range-pair-declaration
        for-range-statement ; for-range-pair-declaration
    **for-range-pair-declaration:**
        **for-range-declaration : for-range-initializer**
    for-range-declaration:
        attribute-specifier-seq(opt) decl-specifier-seq declarator
    for-range-initializer:
        expression
        braced-init-list
```

Change 6.5.4 paragraph 1 to:

```
For a range-based for statement let the iteration-value-declaration be
equivalent to assigning its relative range's iterator value-type for all
ranges __range1 through __rangeN
    for-range-declaration-N = *__beginN;
and for each range-pair-declaration-initializer to be determined as
follows:
    For a range-pair-declaration using the expression form of the
    for-range-initializer
        for-range-declaration : expression
    let range-and-iter-init be equivalent to defining __rangeN to be the
    expression surrounded by parentheses
        auto&& __rangeN = ( expression );
    and for a range-based for statement using the braced-init-list form
    of the for-range-initializer
        for-range-declaration : braced-init-list
    let range-and-iter-init be equivalent defining __rangeN to be the
    braced-init-list
        auto&& __rangeN = braced-init-list;
    in each case let range-and-iter-init also define __beginN and __endN
    to be equivalent to
        auto __beginN = begin-exprN;
        auto __endN = end-exprN;
    where __rangeN, __beginN, and __endN are variables defined for
    exposition for each of the defined ranges only, and _RangeT is
    the type of the expression, and begin-exprN and end-exprN are
    determined as follows:
        (1.1) if _RangeT is an array type, begin-exprN and end-exprN are
        __rangeN and __rangeN + __bound, respectively, where __bound is
        the array bound. If _RangeT is an array of unknown size or an
        array of incomplete type, the program is ill-formed;

        (1.2) if _RangeT is a class type, the unqualified-ids begin and
        end are looked up in the scope of class _RangeT as if by class
        member access lookup (3.4.5), and if either (or both) find at
        least one declaration, begin-exprN and end-exprN are
        __rangeN.begin() and __rangeN.end(), respectively;

        (1.3) otherwise, begin-exprN and end-exprN are begin(__rangeN)
        and end(__rangeN), respectively, where begin and end are looked
        up in the associated namespaces (3.4.2). [Note: Ordinary
        unqualified lookup (3.4.1) is not performed. —end note]

A range-based for statement is equivalent to

{
    range-and-iter-init
    for ( ;
            iterator-continuation-comparison;
            iteration-action ) {
        iteration-value-declaration
        statement
    }
}

where iterator-continuation-comparison is equivalent to comparing
__begin1 through __beginN to its respective __end1 through __endN using
the not equal comparison for each of the ranges declared separated by the
logical AND operator
    __begin1 != __end1 && ... && __beginN != __endN
and iteration-action is equivalent to incrementing __begin1 through
__beginN for each of the ranges declared separated by a comma
    (void)++__begin1, ..., (void)++__beginN

[Example:
int array[5] = { 1, 2, 3, 4, 5 };
for (int& x : array)
    x *= 2;
—end example]

[Example: where ranges are of equal size
int cont1[5] = { 1, 2, 3, 4, 5 };
int cont2[5] = { 6, 7, 8, 9, 10 };
for (auto&& val1 : cont1; auto&& val2 : cont2)
    val1 *= val2;

// expected content of cont1 = { 6, 14, 24, 36, 50 }
—end example]

[Example: where ranges are of differing size
int cont1[5] = { 1, 2, 3, 4, 5 };
int cont2[5] = { 6, 7, 8 };
for (auto&& val1 : cont1; auto&& val2 : cont2)
    val1 *= val2;

// expected content of cont1 = { 6, 14, 24, 4, 5 }
—end example]
```

References

1) [tiny] simultaneous iteration with new-style for syntax, http://cplusplus.github.io/EWG/egg-active.html#43
2) Current discussion for this proposal, https://groups.google.com/a/isocpp.org/forum/m/?utm_source=digest&utm_medium=email#!topic/std-proposals/0Fl4edUN1oo
3) Boost library zip_iterator documentation, http://www.boost.org/doc/libs/iterator/doc/zip_iterator.html
4) Working draft of next C++ specification [N4527] {section 6.5}
5) Current working draft for "Ranges" proposal [N4382]
6) Working example of library implementation [by A. Joël Lamotte (Klaim)], http://melpon.org/wandbox/permlink/gqhKDPZNl0Bp3mMA
7) Current discussion for "Multiple Return Value by tuple conversion", https://groups.google.com/a/isocpp.org/forum/m/?utm_source=digest&utm_medium=email#!topic/std-proposals/jHKDODRNvCU