Proposal of Upto Expressions: [a..b)

Document No.: P0268R0
Date: 2016-02-12
Reply-to: Mike Spertus, Symantec <mike_spertus@symantec.com>
Andrew Tomazos <andrewtomazos@gmail.com>
Audience: Evolution

Summary

We propose a new kind of expression called an *upto expression*:

```
[a..b)
```

It generates the half-open range a, a+1, a+2, ..., b-2, b-1.

Appetizers

```
vector<int> v = [0..5);  // v holds 0 1 2 3 4

for (int i : [0..5)) print(i);  // outputs 0 1 2 3 4
```

Description

An upto expression represents a range of sequential values along some interval specified by its bounds. The syntax is based on the time-honored and intuitive mathematical notation of using a sequence of dots to represent “and so forth”, and for a round or square bracket to represent an exclusive interval bound or inclusive interval bound, respectively.

The expression resulting from an upto expression:

1. has similar properties to a generator like:

```
[] { for (T x = a; x != b; ++x) co_yield x; }
```

2. is compatible with ranges (specifically the type of [a..b) conforms to the View concept).

3. as a range, it is lazily evaluated. (ie The memory usage of [a..b) is proportional to make_pair(a,b), **not** an array of length b-a.)

Motivation

An analysis of the ACTCD16 dataset (containing 2.5 million C/C++ source files taken from 10,000+ of the largest open source projects) revealed the following:

- The identifier i occurred 20,224,291 times.
- The identifier i was the most commonly appearing identifier, by far.
- The identifier i occurred on average 8 times per source file!

We claim the types of ranges describable by upto expressions are an extremely prevalent construct within all C++ programming domains. A “successor sequence” is a foundational entity from discrete mathematics - fundamental throughout not just software engineering but the whole gamut of STEM disciplines. Programmers are reaching for them constantly to compose all sorts of algorithms and data structures.

In particular, indexed loops like

```
for(int i = 0; i < 10; i++) { /* ... */ }
```

are used extensively in C++ code.

By classifying random samples from both ACTCD16 and searchcode.com, we found that over 90% of general for statements (;;) have this form, and of those, only half are candidates for replacement by range-based for. In the remaining half, the loop variable is used in a way other than as the key of a single container - as such are candidates for replacement with an upto expression.

While shorter is not necessarily better, in our opinion,

```
for(int i : [0..10)) { /* ... */ }
```

seems to be more natural and clearly register intent.

As a range expression, [0..10) is also composable within the range ecosystem in a way that for loops, as statements, are not. For example [0..10) can be used as a subexpression, initializer, stored in a variable or passed to or returned from a function.

Examples

Example 1

Integers

```
for (int i : [0..5))
  print(i); // outputs 0 1 2 3 4
```

Example 2

With an upcoming version of ranges you will be able to convert a range to a container:

```
const vector<int> v = [0..5); // v holds 0 1 2 3 4
```

Upto expressions work with iterators:

```
for (auto it : [v.begin()..v.end()))
  print(*it); // outputs 0 1 2 3 4

for (auto it : [v.begin()+2..v.end()-1))
  print(*it); // outputs 2 3
```

Example 3

You can use upto expressions with pointers:

```
char c[] = "foobar";

for (char* p : [c+2..c+5))
  print(*p); // outputs o b a
```

Example 4

Given some user-defined integer-like type Bignum, they will also work (without extending Bignum):

```
Bignum a = /*...*/;
Bignum b = /*...*/;
```

```
for (Bignum i : [a..b))
  print(i); // outputs a, a+1, a+2, ..., b-2, b-1
```

Example 5

Upto expressions are composable with Ranges to enable reversals and strides:

```
using namespace ranges::view;

for (int i : [0..5) | reverse)
  print(i); // outputs 4 3 2 1 0

for (int i : [0..10) | stride(3))
  print(i); // outputs 0 3 6 9

for (int i : [0..10) | stride(3) | reverse)
  print(i); // outputs 9 6 3 0
```

Eric Niebler observed that orthogonally resusing reverse and stride from ranges “make this core facility more powerful while keeping it simple.”

Note that this is efficiently achieved by utilizing the random access traversal operations of the underlying type if available (ie +=3 not ++ 3 times).

Example 6

Like other kinds of expression such as lambdas, generators and ranges - upto expressions can be stored and passed around:

```
auto f() {
  return [0..1’000’000);
}

void g() {
  auto r = f();
  for (int i : r)
    print(i); // outputs 0 1 2 3 4 and so forth up to 999999
  for (int j : r)
    print(j); // outputs 0 1 2 3 4 and so forth up to 999999
}
```

Note that this example uses small constant memory (does not store 1,000,000 ints at any time).

Informal Specification

A new token is introduced to [lex.operators]:

```
    ..
```

A new primary expression is introduced into the grammar:

```
upto-expression:
  [ expression .. expression )
```

Let us call the first subexpression the *begin value* and the second subexpression the *end value*.

The type of the begin value and end value are converted to their common type (as per std::common_type). This common type is known as the *element type*.

The result value of an upto expression has similar properties to range::iota(a,b). It shall conform to the Range View concept. Essentially, it provides a begin() and end() method. The begin() method returns an iterator that contains the begin value as a subobject. The end() method returns an iterator that contains the end value as a subobject. Operations on the iterators are forwarded to their contained values. When the iterators are dereferenced their contained values are returned.

Possible Future Extensions

Extension #1: [a..) means [a..∞)

It is unclear how common we would expect this to be.

In this case the element type would be deduced from the left bound directly.

Extension #2: Add [a..b]

In this extension the roundness or squareness of each bracket specifies whether the range includes (open) or excludes (closed) the bound on the corresponding side.

For example:

```
for (int i : [1..5]) print(i);  // outputs 1 2 3 4 5
```

There are some technical difficulties with supporting closed ranges in a way other than [a..b+1).

By requiring the programmer to provide an exclusive end value that is included in the set of values representable by the type, it ensures that we have a suitable end value (and statically deduced element type) to be contained in the end iterator returned by the .end() function. If this were not the case we could get into infinite wraps:

```
uint8 a = 0;
uint8 b = 255;
for (uint8 i : [a..b] )
   /* oops… infinite loop*/
```

Also, consider what would happen with a would-be closed range like [1..N] when N is zero.

Whereas, with the proposed half-open range like [0..N), we find that the range has N elements, even when N is zero.

Half-open ranges fit better with C++ for a variety of other reasons, not the least of which is consistency with the convention of half-open begin/end ranges of iterators, and fitting more closely with the zero-based indexing of arrays, vectors and other indexing interfaces in the standard library.

Despite of this, we are strongly considering these as a possible future extension.

There are “elaborate ways” to avoid the technical problems, and the extension to this notation is syntactically easy and naturally appeals to the long-standing mathematical notation precedent.

Our recommendation is to keep the design space open to [a..b], while only standardizing the half-open range [a..b) in v1. Once we have experience with [a..b) we can then use this experience to inform our decision whether or not to offer [a..b] (as well as leaving the design space open for the other two exclusive left bound interval notations (a..b] and (a..b) if there turn out to be clear and compelling use cases for them.)

Extension #3: Overloading

We do not propose allowing user-defined overloading of the upto-operator at this time as it would require significant novel core language design work that we view as premature until possible use cases become clear.

Furthermore, the up-to operator already has a customization point in that it can leverage user-defined `operator++()` and `operator!=()` for the enclosed type. Even without any overloading or extension, the bound values of an upto-expression are already accessible as follows:

```
    auto upto_expression = [a..b);
    auto left_bound = *(upto_expression.begin());
    auto right_bound = *(upto_expression.end());
```

One could write a template function to perform this extraction, so upto-expressions could already be given a different meaning when appearing as function arguments:

```
  template<typename UptoExpression>
  R foo(UptoExpression upto_expression) {
    return f(*upto_expression.begin(),*upto_expression.end());
 }
```

Now the function call expression `foo([a..b))` can be endowed with arbitrary meaning.

We do not rule out allowing overloading upto-expressions as a possible future extension as one could imagine that overloading could be useful as in the following fanciful extension to Boost::Spirit:

```
  qi::rule<int()> intParser = *[0..9];
  qi::rule<int()> hexParser = *([0..9] | ['a'..'f']);
```

Overloading would require significant novel and core design work, so we don’t propose this level of overloading being available in version 1. We can take a look after experience with version 1 to decide if there is significant enough demand to warrant such an extension.

Extension #3: Reverse sequences and Different Strides

In our study we found that 7% of traversals use the reverse direction and <1% use a different stride other than ++.

As shown in Example 5 upto expressions are composable with reverse and stride from ranges. Nevertheless, we recognize that this notation is a little cumbersome and confusing for writing for loops that are descending or use non-standard strides and suspect people may continue to use the current approach for those. One nice thing about [a..b) is that it could be extended to support reverse iteration and strides in the future if that proves to be worthwhile. For example (**Note:** we are not proposing this) [5, 4, .. 0) could be an extension for a descending stride.

Syntax Design

We evaluated several different potential syntaxes for upto expressions before arriving at the recommended [a..b) syntax:

Syntax #1: a...b

Pros

- No new token. Reuses ellipsis.
- Communicates “and so forth” semantic.

Cons

- Ellipsis token is already heavily overloaded.
- Compiler heroics required to disambiguate from fold expressions and pack expansions.
- Does not communicate half-openness of bounds. That is, 1...5 may be misread as 1,2,3,4,5 not 1,2,3,4.

Syntax #2: a..b

Pros

- Communicates “and so forth” semantic.
- Precedent in many other languages, both old and new.

Cons

- Does not communicate half-openness of bounds. That is, 1..5 may be misread as 1,2,3,4,5 not 1,2,3,4.
- Requires new token.

Syntax #3: a..<b or a…<b

Pros

- Could unify with N4235
- Communicates “and so forth” semantic.
- Communicates half-openness.
- Might be extensible to reverse a..>b
- Might be extensible to inclusive right bound a..<=b.

Cons

- It's visually similar to a fold expansion (...<x) which means something different x1 < x2 < … < xn. Any follow on paper to N4235 may run into the same issue.
- We feel it does not communicate half-openness as naturally as [a..b)
- It cannot be extended to express inclusion or exclusion of the left-bound.
- In the case of at least iterators, upto expressions might confuse the use of the < token with comparison, whereas != is used. ie for (auto it = v.begin(); it **!=** v.end(); ++it)

Syntax #4: [a, b)

Pros

- Directly appeals verbatim to mathematical interval notation.

Cons

- Not compatible with Extension #4: [a,b] (parser heroics required to disambiguate from lambda introducer) or (a,b) (ambiguous with parenthesized comma-expression).
- Could be misread as a continuous interval. That is [2,4) are all real numbers between 2.0 (inclusive) and 4.0 (inclusive). ie including for example 2.1 and π

Syntax #5: [a..b)

Pros

- Communicates "and so forth" semantic with dots.
- Appeals to the mathematical interval notation of using round or square brackets to represent inclusive/exclusive bounds.
- Able to express inclusive/exclusive on both bounds.
- Is extensible to all the extensions: [a..b], (a..b), (a..b], and [a..)

Cons

- Requires new token.
- Not so good for descending counts. An extension to [5..0) may or may not be OK conceptually - but code generated from [a..b) needs comparison support on the element type and an additional runtime compare-and-branch to select forward or reverse iteration order. This performance penalty may be unacceptable.

Decision

Based on the above considerations, we are proposing Syntax #5.

Wording

In §2.12p1 [lex.operators], modify the definition of *preprocessing-op-or-punc* as follows

preprocessing-op-or-punc: one of

```
{       }       [       ]       #       ##      (       )
<:      :>      <%      %>      %:      %:%:    ;       :       ...
new     delete  ?       ::      .       .*      ..
+       -       *       /       %       ^       &       |       ~
!       =       <       >       +=      -=      *=      /=      %=
^=      &=      |=      <<      >>      >>=     <<=     ==      !=
<=      >=      &&      ||      ++      --      ,       ->*     ->
and     and_eq  bitand  bitor   compl   not     not_eq
or      or_eq   xor     xor_eq
```

In §5.1.1 [expr.prim.general], modify the definition of *primary-expression* as follows

primary-expression:
literal
`this`
`(` *expression* `)`
id-expression
lambda-expression
fold-expression
upto-expression

Between §5.19 [expr.comma] and §5.20 [expr.const], put a new section

upto operator **[expr.upto]**

upto-expression:
`[` *expression* `..` *expression* `)`

The evaluation of an *upto-expression* results in a prvalue temporary (12.2). This temporary is called the *upto object*. The type of the *upto-expression* (which is also the type of the upto object) is a unnamed non-union class type - called the *upto type* - whose properties are described below. The expression preceding the `..` is called the *left-upto-expression*. Its type is called the *left-upto-type*. The type of the expression following the `..` is called the *right-upto-expression*. Its type is called the *right-upto-type*.

The upto type has

- a public member typedef named `value_type` of type `common_type<`*left-upto-type*`,` *right-upto-type*`>::type`
- a public member typedef named `iterator` of type __upto_iterator<value_type>
- a public inline `iterator begin()` method returning __upto_iterator(*left-upto-expression)*
- a public inline `iterator end()` method returning __upto_iterator(*right-upto-expression)*

Where __upto_iterator<T> has a data member of type T and the following members:

- *a constructor from T that copy-constructs the data member*
- *an operator* that returns the data member*
- *an operator++, operator== and operator != that forwards the operation to the data member*
- *for all other operator@, if T supports operator@ so shall __upto_iterator<T>, and each shall forward the operation to the data member.*

[Note: The intention is that the upto type conforms to the Range View concept. -end note]

[Example:
```
 for(int i: [0..5)) {
    cout << i << ", "; // Prints 0, 1, 2, 3, 4,
  }
```
-- end example]