

## New Safer Functions to Advance Iterators

Document number: N4317  
Date: 2014-11-17  
Project: Programming Language C++, Library Evolution Working Group  
Reply-to: Patrick Grace<patrick.grace@ucd.ie>

### I. Introduction

This is a simple proposal to add new safer more robust versions of `advance`, `next`, and `prev` for iterators to the standard C++ library.

### II. Motivation and Scope

On the few occasions that I needed a function that advances iterators I found that the standard library functions weren't safe to use without awkward checking that the iterator wasn't going to overrun the container boundary. (Awkward here could mean doing a distance calculation to the end of the container, which runs in linear time for many iterators.) For example, if I needed to process every 3<sup>rd</sup> element in a container I would like to be able to do something like:

```
for ( auto i = begin(con); i != end(con); advance(i, 3) )
    process( *i );
```

This loop has only a one in three chance of landing exactly on `end(con)`. When it doesn't, it advances into undefined behaviour, possibly crashing. The same problem happens if we use `next`:

```
for ( auto i = begin(con); i != end(con); i = next(i, 3) )
    process( *i );
```

A novice user would probably expect this example to work and we should be trying to make C++ easier to learn. Also, abstract thinkers would probably treat `advance` and `next` as black boxes without giving any thought to their internal behaviour that could fail under certain circumstances.

I think that if these functions were safer to use they would be used more extensively. As it is they are restricted to being used when the distance to the end is known.

### III. Proposal

I propose a new version of `advance`, `next`, and `prev` that each takes an extra parameter:

```
template <class InputIterator>
void advance(InputIterator& i, InputIterator e,
            typename std::iterator_traits<InputIterator>::difference_type n);

template <class ForwardIterator>
InputIterator next(ForwardIterator i, ForwardIterator e,
                 typename std::iterator_traits<ForwardIterator>::difference_type n = 1);

template <class BidirectionalIterator>
BidirectionalIterator prev(BidirectionalIterator i, BidirectionalIterator e,
                          typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);
```

The requirements for `i` and `n` are the same as for the existing functions. The extra parameter, `e`, is required to act as a safe end limit for `i` in the direction that `i` is advancing. If `e` is not in the direction that `i` is advancing the behaviour should be undefined.

For random-access iterators the new version of `advance`, `next`, and `prev` will first check the distance between `i` and `e` and compare it with `n`. If the distance is too large `i` is set equal to `e`, otherwise `i` is incremented by `n` as usual. Although taking slightly longer than the existing functions on account of the extra check this is still achieved in constant time.

For input, forward or bidirectional iterators it will check equality between `i` and `e` after each increment, returning if `e` is reached. Although taking slightly longer than the existing functions on account of the extra `n` checks this is still achieved in linear time.

The above example can now be written like this and will work in all cases:

```
for (auto i = begin(con), e = end(con); i != e; advance(i, e, 3))
    process( *i );
```

or using `next`:

```
for (auto i = begin(con), e = end(con); i != e; i = next(i, e, 3))
    process( *i );
```

or an example with negative `n` (using reverse iterators might be better):

```
auto i = end(con), e = begin(con);
for (--i; i != e; advance(i, e, -3))
    process( *i );
```

## IV. Impact on the Standard

The proposal only requires the addition of the new functions to section 24.4.4 Iterator Operations of the standard as shown below in the technical specifications. It is a pure extension without requiring any other changes. It can be implemented using C++ 11 compilers.

## V. Design Decisions

The original `advance` function in the standard has the type of `n` as a template parameter rather than using the difference type of the iterator. I assume this is for backward compatibility and should not be repeated in new functions. Therefore, I selected the difference type of the iterator as the type for `n`.

I did consider putting the new parameter at the end rather than in the middle. For `advance` this would make sense and might be more intuitive (advance `i` by 3, but stop at `e`). However, for `next` and `prev` this is not possible because of the default `n` value which has to be at the end. It is probably better if all three functions have the same parameter order. This would also fit in with the observed pattern that a lot of algorithms start with a pair of iterators in the same range.

Someone also suggested including completely safe functions that have two extra parameters: both the upper and lower safe limits. For example, for `advance`:

```
template <typename Iter>
void advance( Iter& i, Iter b, Iter e,
             typename std::iterator_traits<Iter>::difference_type n ) {
    if ( n > 0 )
        advance(i, e, n);
    else if ( n < 0 )
        advance(i, b, n);
}
```

I think this is overkill, mainly because I can't think of where I would use it and as shown it is easy to check the sign of *n* before calling the correct function.

## VI. Possible Implementation

The following simple implementation was tested on VC++ 2013 using a variety of containers and also with reverse iterators :

```
template <typename RanIter, typename Dist>
void advance_helper( RanIter& i, RanIter e, Dist n,
                    std::random_access_iterator_tag ) {
    auto maxd = e - i;
    if ( ( n > 0 && maxd < n ) || ( n < 0 && maxd > n ) )
        i = e;
    else
        i += n;
}

template <typename InpIter, typename Dist>
void advance_helper( InpIter& i, InpIter e, Dist n,
                    std::input_iterator_tag ) {
    for ( ; 0 < n && i != e; --n )
        ++i;
}

template <typename BiIter, typename Dist>
void advance_helper( BiIter& i, BiIter e, Dist n,
                    std::bidirectional_iterator_tag ) {
    for ( ; 0 < n && i != e; --n )
        ++i;
    for ( ; n < 0 && i != e; ++n )
        --i;
}

template <typename Iter>
void advance( Iter& i, Iter e,
             typename std::iterator_traits<Iter>::difference_type n ) {
    typedef std::iterator_traits<Iter>::iterator_category Cat;
    advance_helper( i, e, n, Cat{} );
}

template <typename Iter>
Iter next( Iter i, Iter e,
          typename std::iterator_traits<Iter>::difference_type n = 1 ) {
    advance( i, e, n );
    return i;
}

template <typename Iter>
Iter prev( Iter i, Iter e,
          typename std::iterator_traits<Iter>::difference_type n = 1 ) {
    advance( i, e, -n );
    return i;
}
```

As mentioned above, if  $e$  is not the direction that  $i$  is advancing the behaviour is undefined. For this implementation, a random-access iterator will return leaving  $i = e$ , but the other iterators continue to behave as the existing version of `advance`, possibly crashing if a safe limit is overshot.

## VII. Technical Specifications

Amend section 24.4.4 Iterator Operations of the standard to include the new functions as follows (additions in blue):

```
template <class InputIterator, class Distance>
    void advance(InputIterator& i, Distance n);
```

```
template <class InputIterator>
    void advance(InputIterator& i, InputIterator e,
                typename std::iterator_traits<InputIterator>::difference_type n);
```

*Requires:*  $n$  shall be negative only for bidirectional and random access iterators.  $e$  shall be an iterator that is a safe limit for  $i$  in the respective container in the direction of advancement.

*Effects:* Increments (or decrements for negative  $n$ ) iterator reference  $i$  by  $n$ , or stopping at  $e$  if it is specified and reached first.

...

```
template <class ForwardIterator>
    ForwardIterator next(ForwardIterator x,
                       typename std::iterator_traits<ForwardIterator>::difference_type n = 1);
```

*Effects:* Equivalent to `advance(x, n); return x;`

```
template <class ForwardIterator>
    ForwardIterator next(ForwardIterator x, ForwardIterator e,
                       typename std::iterator_traits<ForwardIterator>::difference_type n = 1);
```

*Effects:* Equivalent to `advance(x, e, n); return x;`

```
template <class BidirectionalIterator>
    BidirectionalIterator prev(BidirectionalIterator x,
                              typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);
```

*Effects:* Equivalent to `advance(x, -n); return x;`

```
template <class BidirectionalIterator>
    BidirectionalIterator prev(BidirectionalIterator x, BidirectionalIterator e,
                              typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);
```

*Effects:* Equivalent to `advance(x, e, -n); return x;`

## VIII. Acknowledgements

Thanks to Stephan T. Lavavej for reviewing an earlier version of this document and giving valuable feedback.