Document number: N4128
Date: 2014-10-10
Project: Programming Language C++, Library Working Group
Reply-to: Eric Niebler <eniebler@boost.org>,
Sean Parent <sparent@adobe.com>,
Andrew Sutton
<andrew.n.sutton@gmail.com>

Ranges for the Standard Library, Revision 1

> “A beginning is the time for taking the most delicate care that the balances are correct.”
>
> – Frank Herbert, *Dune*

1 Introduction

This paper outlines what support for ranges in the C++ standard library might look like. Rather than presenting a final design, this paper proposes a set of concepts and guidelines for using them to implement range-based versions of the standard algorithms. It draws inspiration from the Boost.Range[4] library, the range algorithms in Adobe Source Libraries[1], *Elements of Programming* by Stepanov and McJones (2009) [20], and from N3351 “A Concept Design for the STL” by Stroustrup and Sutton (2012) [21]. In addition to presenting the concepts and guidelines, this paper discusses the rationale behind each, weighing different design options.

The decision to defer any discussion about specific wording was taken in recognition of the fact that any range design is likely to undergo significant revision by the committee. The paper is intended merely as a starting point for discussion and as a basis for future work.

This paper assumes the availability of Concepts Lite; however, everything suggested here has been implemented in C++11, where Concepts Lite has been simulated with the help of generalized SFINAE for expressions.

2 Motivation and Scope

A *range* is an object that refers to a sequence of elements, conceptually similar to a pair of iterators. One prime motivation for ranges is to give users a simpler syntax for calling algorithms. Rather than this:

```
std::vector<int> v { /*...*/ };
std::sort( v.begin(), v.end() );
```

Ranges would give us a pithier syntax:

```
std::sort( v );
```

Allowing algorithms to take a single range object instead of separate begin and end iterators brings other benefits besides convenience. In particular:

- It eliminates the possibility of mismatched iterators.
- It opens the door to *range adaptors* which lazily transform or filter their underlying sequence in interesting ways.

Range adaptors are far more compelling than iterator adaptors due to the fact that only a single object, the range object, needs to be adapted; hence, adaptors can be easily chained to create lazy computational pipelines, as in the code below which sums the first 10 squares:

```
int total = accumulate(view::iota(1) |
                       view::transform([](int x){return x*x;}) |
                       view::take(10), 0);
```

The standard defines the term “range” in [iterator.requirements.general]:

> […] in general, a range `[i,j)` refers to the elements in the data structure starting with the element pointed to by `i` and up to but not including the element pointed to by `j`. Range `[i,j)` is valid if and only if `j` is reachable from `i`.

From the perspective of the standard library, a range *is* a pair of iterators. But there are other interesting ways to denote a range of elements:

- An iterator and a count of elements
- An iterator and a (possibly stateful) predicate that indicates when the range is exhausted.

One of these three range types can be used to denote all other range types, like an iterator and a sentinel value (e.g. a null-terminated string), or a range that spans disjoint ranges. Ideally, we would like our Range abstraction to be general enough to accommodate the three different kinds of ranges since that would increase the applicability of the algorithms.

2.1 Impact on the Standard

Although this paper does not offer specific wording for any additions to the standard, we imagine that proper support for ranges in C++ would involve changes to the following parts of the standard:

- New library-wide concepts related to ranges.
- New iterator algorithms for efficiently dealing with the new abstractions.
- Changes to existing algorithms to constrain the templates with concepts.
- Additional overloads of existing algorithms that accept ranges instead of pairs of iterators.
- Changes to the containers to allow containers to be constructed and assigned from ranges, and to allow range-based insert operations.
- A new library section for range adaptors, which are views of existing data that have been transformed or filtered and that compose with other views.
- General utilities for the construction of custom range adaptors.
- A minor change to the specification of the range-based `for` to make it more efficient and general.

Future papers will make specific recommendations for all of the above, modulo any feedback on the design presented here.

3 Proposed Design

The design space for ranges is surprisingly large. At one end of the spectrum lies Boost.Range[4] and Adobe's ASL[1] in which ranges are a thin abstraction on top of iterators, which remain the primitives that glue together data structures and algorithms. At the other end of the spectrum we find the D Standard Library's std.range module[6], in which ranges and operations on them are the primitives themselves.

This proposal picks a single point in this design space, and here we present the decisions that led to the selection of that point, along with guidelines to be applied to the standard library and the rationale for each choice.

3.1 Design Goals

We feel that a well-designed range abstraction would:

- Allow algorithms to operate on the three kinds of ranges with low or no abstraction penalty and a minimum of syntactic noise,
- Allow range-based algorithms to share implementation with iterator-based algorithms,
- Make it easy for users to reason about the complexity and expense of range operations (e.g. How many passes over the data are made? Are the elements copied? etc.),
- Protect the user from lifetime issues,
- Make it straightforward for users to make their types model one of the range concepts.

It is helpful at this point to reflect on the success of C++11's range-based `for` loop. It succeeds because most of the types over which one would want to iterate already define iterators and `begin`/`end` members. Cleanly and efficiently interoperating with and reusing the existing abstractions of the STL is critical to the success of any range extensions.

3.2 High-Level Design

At the highest level, this paper proposes the addition of two related range concepts: Iterable and Range. An *Iterable* type is one for which we can call `begin()` and `end()` to yield an iterator/sentinel pair. (Sentinels are described below.) The Iterable concept says nothing about the type's constructibility or assignability. Range-based standard algorithms are constrained using the Iterable concept. Consider:

```
int buf[5000];
// Fill buf
std::sort( buf );
```

`buf` denotes a random access range of elements, so we should be able to sort it; but native arrays are neither copyable nor assignable, so these operations should not be required by whatever range-like concept is used to constrain `sort`. The above line of code is equivalent to:

```
using std::begin;
using std::end;
std::sort( begin( buf ), end( buf ) );
```

For an Iterable object `o`, the concept requires the following:

```
auto b = begin( o ); // b models Iterator
auto e = end( o );   // e models Regular
bool f = (b == e);   // b and e model EqualityComparable
```

Algorithms will typically be implemented to take iterator/sentinel pairs, rather than the iterator/iterator pairs as they do now. A typical algorithm might look like:

```
template<Iterator I, Regular S, /*...*/>
    requires EqualityComparable<I, S>
I some_algo(I first, S last, /*...*/)
{
    for(; first != last; ++first)
        /*...*/
    return first;
}

template<Iterable R, /*...*/>
IteratorOf<R> some_algo( R & r, /*...*/ )
{
    return some_algo( begin(r), end(r), /*...*/ );
}
```

The *Range* concept is modeled by lightweight objects that denote a range of elements they do not own. A pair of iterators can be a model of Range, whereas a `vector` is not. Range, as opposed to Iterable, requires copyability and assignability. Copying and assignment are required to execute in constant time; that is, the cost of these operations is not proportional to the number of elements in the Range.

The Range concept refines the Iterable concept by additionally requiring following valid expressions for an object `o` of type `O`:

```
// Constructible:
auto o1 = o;
auto o2 = std::move(o);
O o3; // default-constructed, singular
// Assignable:
o2 = o1;
o2 = std::move(o1);
// Destructible
o.~O();
```

The Range concept exists to give the range adaptors consistent and predictable semantics, and memory and performance characteristics. Since adaptors allow the composition of range objects, those objects must be efficiently copyable (or at least movable). The result of adapting a Range is a Range. The result of adapting a container is also a Range; the container – or any Iterable that is not already a Range – is first converted to a Range automatically by taking the container's `begin` and `end`.

The precise definitions of the suggested concepts are given in Section 4, along with other supporting concepts that have proven useful while porting the algorithms.

The use of sentinels instead of iterators as an Iterable's bound is best understood by seeing how the three different kinds of ranges can be made to model the Iterable concept. Below is a sketch of how `begin` and `end` might work for each kind.

- **Pair of iterators**: An end iterator is a perfectly acceptable sentinel. Existing code that uses iterator pairs to call STL algorithms will continue working with no changes.
- **Iterator and predicate**: `begin(rng)` can return a normal iterator, `first`. `end(rng)` can return a sentinel `last` such that `first == last` returns the result of calling `last.predicate_(*first)`. See Appendix 1 for a discussion about the code generation benefits of letting the sentinel have a different type than the iterator.
- **Iterator and count**: `begin(rng)` can return an iterator `first` that bundles the underlying iterator with the count to the end. Incrementing the iterator decrements the count. `end(rng)` can return an empty sentinel `last` such that `first == last` returns the result of `first.count_ == 0`. See Appendix 4 for a discussion of the performance implications of this design.

3.3 Design Decisions, Guidelines, and Rationale

Below we present the decisions that led to the chosen high-level design, along with guidelines to be applied to the standard library and the rationale for each choice.

3.3.1 Iterator Operations are Primitive

The most fundamental decision facing the designer of a generic library like the STL is: what are the *basis operations*? Basis operations are the primitive operations upon which all other desired operations can be efficiently built. In the STL, those operations are the operations on iterators, and they are clustered into the familiar iterator concept hierarchy. Two iterators are required to denote a range. Can these two positions be bundled together into a single range object and – more ambitiously – can operations on ranges be made the basis, obviating the need for iterators entirely?

Below we describe two libraries that use range operations as the basis, and describe why they are not a good fit for the C++ Standard Library.

3.3.1.1 D's Ranges

In the C++ Standard Library, iterators fill several roles:

- Two of them denote a sequence of elements.
- One of them denotes a position within a range.
- They allow access to an element at the current position.
- They allow access to subsequent (and sometimes prior) positions in the sequence.

The D Standard Library[6] takes a different approach. In D, ranges and the operations on them form the basis operations of the standard algorithms. D-style ranges fill the following roles:

- They denote a sequence of elements.
- They allow access to the front of the range, and sometimes to the back or the N-th.
- They allow the removal of elements from the front of the range, and sometimes from the back.

Would C++ benefit from a similar design? The argument typically given in favor of D's ranges is that they lead to simpler code, both for the algorithms that operate on ranges as well as for users who wish to create custom range types. Code that manipulates positions directly can be harder to reason about and thus more bug-prone, and implementing custom iterators is famously complicated.

D-style ranges can only ever shrink, and they have no notion of position within sequence. If one were to try to implement C++'s iterators on top of

D's ranges, one would immediately run into trouble implementing ForwardIterator's `operator==`. As D's ranges do not represent position, there would be no way to test two ranges to see if their heads referred to the same element. (The `front` member function that returns the front of a D range is not required to return a reference, nor would that be sufficient to implement a hypothetical `hasSameFront` function; a repeating range might return the same element multiple times, leading to false positives.) Additionally, there would be trouble implementing BidirectionalIterator's `operator--` or RandomAccessIterator's `operator+=` as that might require a range to grow, and D ranges can't grow.

On the other hand, two C++ iterators can easily be used to implement a D-style range; thus, every range-based design can be implemented in terms of iterators. Since iterators can implement D ranges, but D ranges cannot be used to implement iterators, we conclude that iterators form a more powerful and foundational basis.

D avoids the limits of its ranges by carefully designing the algorithms such that the missing functionality is never needed. This can sometimes require some creativity, and leads to some awkward productions. A good example is the `find` family of algorithms. Since `find` cannot return the position of the found element, it must instead return a range. But which range to return? The D Standard Library has as many `find` algorithms as there are answers to this question (find, findSkip, findSplit, findSplitBefore, findSplitAfter). All these algorithms find an element in a range; they only differ in the information they return. In contrast, C++ has just one `find` algorithm; it returns a position, and the user is free to construct any desired range from that.

Appendix 3 contains an example of an algorithm that cannot be implemented with the same algorithmic complexity using D-style ranges. Any algorithm that needs to freely move an iterator forward and backward between two bounds will suffer from the same fundamental problem. Just writing the signature of an algorithm like `rotate`, which takes as an argument a position in a sequence, is challenging without a way to specify a position.

Since C++ iterators cannot be implemented on top of D's ranges, iterators have to stay both for the increased expressive power and for backwards compatibility. To additionally provide D-style ranges – essentially offering an incompatible set of basis operations – would create a schism within the standard library between range-based and iterator-based algorithms, which couldn't share code. We, the authors, consider such a schism unacceptable.

3.3.1.2 Position-Based Ranges

If a range-first design that abandons "position" as a representable entity is undesirable for C++, perhaps adding position back in yields a satisfactory design. That is the approach taken by James Touton's range library[16], where ranges – together with a new Position concept – are the primitives. A Position, as its name suggests, represents a position in a range. Unlike an

iterator, a position cannot be used to access the element at that position without the range into which it refers, nor can the position be advanced without the range.

This design has the following advantages:

- In making position a representable entity, it avoids the sometimes awkward constructions of D's range library.
- In requiring the range in order to dereference the position, it avoids all dangling iterator issues.
- In requiring the range in order to change the position, it makes range-checking trivial. This is a boon not just for debuggability, but also for the design of certain range adaptors like filter and stride whose iterators need to know the end of the range so as not to walk past it.
- It permits a clean separation of element traversal and access, much like the suggested cursor/property map abstraction[8].

It is possible to implement iterators on top of a position-based range by bundling a position with a pointer to a range into an iterator. However that results in iterators that may be fatter than necessary.

A more workable approach would be to reimplement all the algorithms in terms of the position-based range primitives, and have the iterator-based overloads (that are retained for backwards-compatibility) forward to the range-based versions. In such a scheme, two iterators can be turned into a position-based range by wrapping them in a single range object and using the iterators themselves as the "positions".

Although workable in principle, in practice it means there will be two ways of representing a range: an object with `begin()` and `end()` members that return iterators, and one with `begin_pos()` and `end_pos()` members that return positions; and every function that accepts ranges will need to account for that. It would mean that everybody would need to learn a new way to write algorithms. And it would mean that sometimes algorithms would return iterators and sometimes they would return positions, and that users would need to learn a new way to access the elements of a range.

As appealing as the design is, it is too disruptive a change for the Standard Library.

3.3.1.3 Basis-Operations: Summary

In short, it just doesn't seem worth the trouble to change basis-operation horses in midstream. Iterators have a proven track record as a solid basis for the algorithms, and the C++ community has a heavy investment in the abstraction. The most heavily used and vetted C++ range libraries, Boost.Range and ASL Ranges, are built on top of iterators and have shown that it works in practice. This proposal follows suit.

3.3.2 Ranges Cannot Own Elements

As described above, a Container is not a Range; it is, however, an Iterable. Distinguishing between the two makes it possible to be explicit about where copyability is required, and with what performance characteristics.

The algorithms discussed in this proposal don't require any distinction between Ranges and Containers since they never copy or assign the ranges passed to them; they only request the `begin` and `end` iterators. The distinction between Iterables and Ranges only becomes important when defining adaptor chains. What does code like the following mean?

```
auto rng = v | view::reverse;
```

This creates a view of `v` that iterates in reverse order. Now: is `rng` copyable, and if so, how expensive is the copy operation? If `v` is a `vector`, can `rng` safely outlive `v`? How about if `v` is just a pair of iterators? What happens when a user does `*rng.begin() = 42`? Is `v` mutated? How do the answers change if we replaced `v` with an rvalue expression? If a copy of `rng` is made, and an element is mutated through the copy, does the original `rng` object "see" the change?

By specifying that Ranges do *not* own their elements, and further specifying that range adaptors operate on and produce Ranges (not Containers), we are able to answer these questions in a clear and consistent way. The result of a chain of range adaptors is always a lightweight object that is cheap to copy and assign (O(1) as opposed to O(N)), and that refers to elements whose lifetime is managed by some other object. Mutating an element through the resulting Range object mutates the underlying sequence. Copies of the resulting range are aliases to the same elements, and mutations to the elements are visible through all the aliased ranges.

If `v` is a Range in the above line of code, it is copied into the adaptor. If it is a `vector`, the `vector` is first used to construct a Range by taking the `begin` and `end` of the Container. This happens automatically.

The downside of this design is that it is sometimes desirable to do this:

```
// Try to adapt an rvalue container
auto rng = vector<int>{1,2,3,4} | view::reverse; // OK?
```

Adaptors operate on and yield Ranges; other Iterables (i.e., containers) are used to construct Ranges by first taking their begin and end. The code above is unsafe because `rng` will be left holding invalid iterators into a container that no longer exists. Our solution is to disallow the above code. *It is illegal to adapt an rvalue non-Range.* (Adapting rvalue Ranges, however, is perfectly acceptable; indeed necessary if adaptor pipelines are to work.) See Appendix 6 for the mechanics of how we distinguish between Iterables and Ranges.

The alternative is for the rvalue container to be moved (or copied) into the adapted range and held by value. The resulting object would therefore no longer model Range; it would model Iterable. The authors feel that this weakening of the requirements on the return type makes it difficult to reason about the semantics and algorithmic complexity of range adaptors. The recommendation is to first declare the container and then create the adaptor separately.

3.3.3 Ranges Are Semiregular

We've already decided that Ranges (not Iterables) are copyable and assignable. They are, in the terminology of EoP[20] and N3351[21], Semiregular types. It follows that copies are independent, even though the copies are both aliases of the same underlying elements. The ranges are independent in the same way that a copy of a pointer or an iterator is independent from the original. Likewise, iterators from two ranges that are copies of each other are also independent. When the source range goes out of scope, it does not invalidate an iterator into the destination range.

Semiregular also requires DefaultConstructible in N3351. We follow suit and require all Ranges to be DefaultConstructible. Although this complicates the implementation of some range types, it has proven useful in practice, so we have kept this requirement.

It is tempting to make Ranges Regular by requiring that they be EqualityComparable. A Range type would satisfy this additional requirement by testing whether two Ranges refer to the same elements in a sequence. (Note that it would be an error for `operator==` to test corresponding elements in two Ranges for equality, in the same way that it would be an error for `operator==` invoked on two pointers to compare the pointed-to elements. The state of a Range is not influenced by the content of its elements; a Range is defined only by the identity of the elements it references.) Although such a requirement is appealing in theory, it has problems:

- It might conflict with users' expectations of what `rng1 == rng2` means. (See string_view for an example of a Range-like class that implements `operator==` in terms of the values of its elements, rather than identity.)
- It is impossible to implement with those semantics in O(1) for some range types; for example, a filter range that stores a predicate. Functors and lambdas are generally not EqualityComparable.

Another option is to allow Ranges to trivially model EqualityComparable by narrowly defining the domain over which the operation is valid. Iterators may only be compared if they refer into the same range. We can extend the reasoning to Ranges, which are logically little more than pairs of iterators. Taking this tack, we could allow a Range type to define its `operator==` as:

```
rng1.begin() == rng2.begin() && rng1.end() == rng2.end()
```

The assumption being that the operation is invalid if `rng1` and `rng2` refer to different elements. Although principled (for some set of principles), such a definition is almost certain to lead users into undefined behavior-land.

As a result of the above, we have decided that the Range concept should not require EqualityComparable. Ranges are Semiregular, not Regular.

If a user would like to check to see if two ranges have elements that compare equal, we suggest the `equal` algorithm:

```
if(std::equal(rng1, rng2))
    // ...
```

3.3.4 Range Iterators Cannot Outlive Their Ranges

Containers own their elements, so it is clear that the container must outlive the iterators it generates. But is the same true for a Range if it does not own its elements? For a trivial range consisting of two iterators, it's clearly not true; the iterators *may* outlive the range.

It turns out that if we require that all range's iterators be permitted to outlive the range, a great many interesting range types become significantly more expensive at runtime. A good case study is the filter view.

A filter view takes a range and a predicate, and presents a view of the sequence that skips the elements for which the predicate is false. (The filter view can be thought of as a lazy equivalent of the `copy_if` algorithm.) The existence of the `boost::filter_iterator` shows that such an iterator *can* be made such that it doesn't depend on a range, but at a cost. The `filter_iterator` stores:

1. An iterator that indicates the current position in the underlying sequence.
2. An iterator that indicates the end of the underlying sequence (needed by the increment operators to avoid falling off the end while searching for an element that satisfies the predicate).
3. The predicate.

In today's STL, the begin and end iterators must have the same type, and they are both needed to call an algorithm. Thus, the information in (2) and (3) is duplicated. Also, the predicate may be expensive to copy, given the ease with which capture-by-value lambdas and `std::functions` can be created. When such iterators are composed with other kinds of views (e.g., a transformed, filtered view), the bloat compounds exponentially (see Index-Based Ranges[19]).

By relaxing the constraint that a range's begin and end must have the same type, we can avoid the duplication, but the begin iterator still must hold everything, which is potentially expensive.

If we could rely on the range object outliving the iterator, we can make the filter iterators smaller and lighter. The range object can hold the predicate and the underlying range's begin/end iterators. The filter view's iterator only needs to hold the current position and a pointer back to the range object.

The same logic applies to the transform view, which applies a transformation function to elements on the fly. If the iterators are required to be valid beyond the lifetime of the transform range, then the transformation function must be cached inside each iterator. This could make the iterators expensive to copy. An implementor might instead want the freedom to put the transformation function in the range object.

A final example is `istream_iterator<T>`. Every `istream_iterator<T>` holds a single cached `T` object inside it. If `T` is an expensive-to-copy type like `std::string`, then copying `istream_iterator`s is potentially causing dynamic allocations. The STL assumes iterators are cheap to copy and copies them freely. A better design would see the cached `string` object move into an `istream_range` object whose iterators merely stored pointers back to the range. This would make the iterators smaller and cheaper to copy, but they would become invalid once the range was destroyed. This tradeoff is probably worth it.

3.3.5 An Iterable's End May Have a Different Type Than Its Begin

In today's STL, `c.begin()` must have the same type as `c.end()`. This is because the only kind of range the STL supports is a pair of iterators. However, we've given examples of other kinds of ranges we would like to support, such as an iterator and a predicate, and an iterator and a count. These kinds of ranges can already be supported by shoe-horning them into the pair-of-iterators mold, but at a cost (see Appendix 1). Loosening the constraints of the type of Iterable's end makes it possible to accommodate these other kinds of ranges with lower overhead.

Allowing "end-ness" to be encoded in the type system also eliminates the need to mock-up dummy end iterators like `std::istream_iterator` and `std::regex_iterator`, the logic of which is tricky to get right. What's more, it improves code generation for these kinds of ranges. With the use of dummy end iterators, information which is known at compile-time – namely, that an iterator represents the end – must be encoded into runtime information in the iterator itself. This robs the compiler of the information it needs to eliminate branches from the core loops of many algorithms. See Appendix 1 for an example of how sentinels can positively affect code generation.

When considering this choice for the range concept, it's helpful to think about how it would affect the algorithms. Consider `std::for_each`, which currently has this signature:

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f)
```

```
{
    for(; first ≠ last; ++first)
        f(*first);
    return f;
}
```

With sentinels, `for_each` might look like this:

```
template<InputIterator I, Regular S, Function<ValueType<I>> F>
    requires EqualityComparable<I, S>
F for_each(I first, S last, F f)
{
    for(; first ≠ last; ++first)
        f(*first);
    return f;
}
```

None of the code in the algorithm had to change. No calling code would have to change either; this is a strictly backwards-compatible change. You might think that this opens a new category of programming errors where developers inadvertently pass mismatched iterator/sentinel pairs. However, this algorithm signature is constrained with concept checks that ensures that `I` and `S` satisfied the cross-type EqualityComparable concept (see N3351[21]). See Appendix 2 for further discussion about iterator/sentinel cross-type EqualityComparability constraint.

To see the benefit of this design, imagine a sentinel type `null_sentinel`:

```
// For determining whether an iterator refers to a null value:
struct null_sentinel
{
    template<Iterator I>
    friend bool operator==(I i, null_sentinel) { return 0 == *i; }
    // ... and friends
};

template<Iterator I>
struct common_type<I, null_sentinel>
   ... see Appendix 2 ...
```

Now we can use `std::for_each` on null-terminated strings without needing to know the length of the string:

```
std::for_each(argv[1], null_sentinel(), f);
```

Of course, all the algorithms would have overloads that also accept range arguments, so this can be further simplified to:

```
std::for_each(null_terminated(argv[1]), f);
```

where `null_terminated(InputIterator)` returns a range `r` such that the `std::end(r)` is a `null_sentinel`.

3.3.5.1 Sentinels and Early Algorithm Termination

One excuse sometimes given for not using the standard algorithm is that they don't give the users a way to break out of them early. The use of sentinels makes that possible. Consider a sentinel constructed from both an end iterator and a predicate. Such a sentinel would compare equal to an iterator *either* when the iterator equals the end iterator *or* when the predicate evaluates to true. Using such a sentinel has the effect of terminating an algorithm early. For instance:

```
// Process work items in a queue, allowing for a user interrupt
std::queue<Work> q;
std::function<void(Work const &)> user_interrupt = /*...*/;
std::for_each( q | view::until(user_interrupt), f );
```

In the above, `view::until` is a range modifier that adds `user_interrupt` as an extra termination condition.

3.3.6 Algorithm Return Types are Changed to Accommodate Sentinels

Notice that in the due course of evaluating `std::for_each` with `null_sentinel` above, the position of the null terminator is found. This is potentially useful information that can easily be returned to the user. It is, in fact, a far more interesting and useful result that the `Function` that `for_each` currently returns. So a better signature for `for_each` should look like this:

```
// Returns an InputIterator i such that (i == last) is true:
template<InputIterator I, Regular S, Function<ValueType<I>> F>
    requires EqualityComparable<I, S>
I for_each(I first, S last, F f);
```

In similar fashion, most algorithm get new return types when they are generalized to support sentinels. This is a source-breaking change in many cases. In some cases, like `for_each`, the change is unlikely to be very disruptive. In other cases it may be more so. Merely accepting the breakage is clearly not acceptable. We can imagine three ways to mitigate the problem:

1. Only change the return type when the types of the iterator and the sentinel differ. This leads to a slightly more complicated interface that may confuse users. It also greatly complicates generic code, which would need metaprogramming logic just to use the result of calling some algorithms. For this reason, this possibility is not explored here.

2. Make the new return type of the algorithms implicitly convertible to the old return type. Consider `copy`, which currently returns the ending position of the output iterator. When changed to accommodate sentinels, the return type would be changed to something like `pair<I, O>`; that is, a pair of the input and output iterators. Instead of returning a `pair`, we could return a kind of pair that is implicitly convertible to its second argument. This avoids breakage in some, but not all, scenarios. This subterfuge is unlikely to go completely unnoticed.

3. Deliver the new standard library in a separate namespace that users must opt into. In that case, no code is broken until the user explicitly ports their code. The user would have to accommodate the changed return types then. An automated upgrade tool similar to clang modernize[5] can greatly help here.

We, the authors, prefer (3). Our expectation is that the addition of concepts will occasion a rewrite of the STL to properly concept-ify it. The experience with C++0x Concepts taught us that baking concepts into the STL in a purely backward-compatible way is hard and leads to an unsatisfactory design with a proliferation of meaningless, purely syntactic concepts. The spirit of N3351[21] is to conceptify the STL in a meaningful way, even at the expense of minor breakage. Although the issue has not yet been discussed, one likely solution would be to deliver a new STL in a separate namespace. Once that happens, it opens the door for other minor breaking changes, provided the benefits are deemed worthy.

3.3.7 Orthogonality of Traversal and Access Is Not Surfaced in the Iterator Concepts

The current iterator concept hierarchy ties together the traversal and access properties of iterators. For instance, no forward iterator may return an rvalue proxy when it is dereferenced; the ForwardIterator concept requires that unary `operator*` return an lvalue. There is no room in the hierarchy for, say, a random-access iterator that returns proxies.

This problem is not new to ranges; however, it has serious consequnces for lazy ranges that apply transformations to elements on the fly. If the transformation function does not return an lvalue, the range's iterator can model no concept stronger than InputIterator, even if the resulting iterator could in theory allow random access. The result in practice is that most range adaptors today have the unfortunate effect of degrading the underlying range's category to Input, thereby limiting the number of algorithms it can be passed to – often for no good reason.

Prior work has been done by Abrahams et. al.[13] to separate the traversal and access properties of iterators in a way that is backwards compatible. When formalizing the iterator concepts for a range library, should new iterator concepts be used, or should we hew to the old, simpler concept hierarchy with its known limitations?

A close reading of the iterator concepts proposed by N3351[21] shows that the problematic requirement on ForwardIterator's reference type has been dropped. That is, if the concepts proposed by Stroustrup, Sutton, et. al., are adopted, there will be nothing wrong with, say, a random-access iterator with a proxy for a reference type. The problems described above go away without any need for further complication of the iterator refinement hierarchy. It's

unclear at this time if that was the authors' intent, and if so what impact it has on the standard algorithms. This area requires more research.

It is noted here that this proposal adopts the N3351[21] iterator concepts as-is.

3.3.8 Additional Overloads of the Algorithms

As should be obvious, this range proposal recommends adding additional overloads of the existing algorithms to allow them to work directly on Iterables. This is done in accordance with the following suggested guidelines:

- Any algorithm that currently operates on a range denoted by two iterators gets an overload where the two iterator arguments are replaced with a single Iterable argument. (NOTE: This does *not* include the counted algorithms like `copy_n` that take an iterator and a count instead of two iterators.)
- All overloads of an algorithm, whether they take Iterables or separate iterator/sentinel arguments, are *semantically identical*. All overloads have the same return type. All evaluate eagerly. The intention is that the Iterable-based overloads delegate to the iterator-based ones and have the the same semantics and algorithmic complexity.

- As described above, algorithms that necessarily process their entire input sequence return the iterator position at the end in addition to whatever else they return. The purpose is to return potentially useful information that is computed as a side-effect of the normal execution of the algorithm. Exceptions to this design guideline are made when one of the following is true:

    - The algorithm might in some cases not consume the entire input sequence. (The point of this exception is to avoid forcing the algorithm to compute something that is not necessary for successful completion. For example, `find`.)
    - When the sole purpose of the algorithm is specifically to compute a single value; hence, changing the return type will necessarily break code using the C++11 version. Examples include `is_sorted` and `accumulate`.

- "Three-legged" iterator-based algorithms (i.e. algorithms that operate on two ranges, the second of which is specified by only a single iterator and is assumed to be long enough) now have 4 versions:

    1. The old three-legged iterator version,
    2. A four-legged version that uses the sentinel of the second sequence as an additional termination condition,
    3. A version that takes an Iterable and an Iterator (which dispatches to the three-legged iterator-based version),
    and

4. A version that takes two Iterables (which dispatches to the four-legged iterator-based version).

Note: Purely as an implementation consideration, overloads (3) and (4) above must be coded to avoid ambiguity when a native array is passed as the second parameter (where either an Iterable or an Iterator may appear). Arrays are Iterables, but if (3) is naively coded to take an Iterator by value, a native array would also match, since native arrays decay into pointers.

- If an algorithm returns an iterator into an Iterable argument, the Iterable must be an lvalue. This is to avoid returning an iterator that is immediately made invalid. Conversely, if no iterator into an Iterable argument is returned, then the Iterable should be taken by forwarding reference (aka "universal reference"[22]).
- Algorithms that do not mutate their input sequence must also work when braced initializer lists [dcl.init] are used in place of Iterables. This can require additional `initializer_list` overloads since a type cannot be deduced from a *braced-init-list* used as an argument.

3.3.9 Range-based for Loop is Changed to Accommodate Sentinels

The current range-based `for` loop assumes that a range's end has the same type as its begin. This restriction can be relaxed to allow range-based `for` to operate on Iterables.

The range-based for loop is currently specified as:

```
{
    auto && __range = range-init;
    for ( auto __begin = begin-expr,
               __end = end-expr;
          __begin ≠ __end;
          ++__begin ) {
        for-range-declaration = *__begin;
        statement
    }
}
```

To accommodate Iterables, the change is as simple as:

```
{
    auto && __range = range-init;
    auto __begin = begin-expr;
    auto __end = end-expr;
    for ( ; __begin ≠ __end; ++__begin ) {
        for-range-declaration = *__begin;
        statement
    }
}
```

This is the only core language change required to fully support Iterables.

3.3.10 Allow Mutable-Only Iterables

If a cv-unqualified type `T` models Iterable, then the type `T const` need not. This permits ranges that maintain mutable internal state such as an `istream_range`.

Consider the performance short-comings of `istream_iterator<string>`. The iterator reads a string from an `istream` and must store it internally so it can be returned from `operator*`. This means that copying an `istream_iterator<string>` probably incurs a dynamic allocation. Copying iterators is not supposed to be expensive.

An alternative range-based design would be to define an `istream_range` class template:

```
template<class T>
class istream_range {
   T value_;
   istream * postr_;
public:
    class iterator {
        istream_range * prng_;
    public:
        /*...*/
    };
    iterator begin() {/*...*/}
    iterator end() {/*...*/}
};
```

In this design, the cached value lives in the range object, not in the iterator. The iterator merely stores a pointer back to the range. As the iterator is advanced, values are read from the stream and stored in the cache. Since the range object is mutated as it is iterated, it would be a lie to provide const overloads of `begin()` and `end()`.

The need for a range that is mutated in the course of iteration also came up in the design of a `view::flatten` adaptor that takes a range of ranges, and turns it into a single range. This adapted range also must mutate an internal cache. The adaptor is used to make Ranges monads, and is used in the implementation of Range Comprehensions[17], which are akin to Python's and Haskell's List Comprehensions. (That discussion is beyond the scope of this document.)

3.3.11 Range Adaptors are Lazy Algorithms

Consider the example given at the start of paper:

```
int total = accumulate(view::iota(1) |
                       view::transform([](int x){return x*x;}) |
                       view::take(10), 0);
```

The semantics of the adaptors (the things in the `view::` namespace) are such that computations are only done on demand, when the resulting adapted range is iterated. In this case, the range pipeline expression does no work aside from setting up a computation. Only as `accumulate` executes is the sequence of integers generated and transformed.

All adaptors have these lazy semantics. This gives users the ability to reason about the algorithmic complexity of their programs. If an underlying range is transformed and filtered and then passed to an algorithm, users can be certain that the sequence will be traversed exactly once.

3.3.12 All Algorithms Accept Sentinels Even If They Need An End Iterator

Some algorithms like `reverse` really need an end iterator to do their job. One option would be to require users to pass an actual end iterator instead of a non-iterator sentinel. The other option is to accept a sentinel and do an O(N) probe for the end first. We opted for the latter option. No such algorithms have complexity better than O(N), so such a probe doesn't affect the overall algorithmic complexity of the algorithm. And it saves the user the hassle of having to find the end herself before calling the algorithm. In short, it should be possible to `reverse` a null-terminated string *without* needing to call `strlen` first.

3.3.13 There Is No Function Signature To Express Does-Not-Mutate-The-Range-Elements

Raw pointers have an advantage over iterators with regard to the const-correctness of function interfaces; namely, that you can use `const` to guarantee that the function will not mutate the pointed-to data. Consider:

```
// This will not mutate the data
template<typename T>
void some_function(const T * data, std::size_t count);
```

When we move to iterators, we lose that expressiveness:

```
// Will this mutate the data?
template<typename Iter>
void some_function(Iter first, Iter last);
```

Since the suggested range design is built on top of iterators, it inherits this limitation. Consider what happens when we change to a range-based API with a simple `range` type like that proposed by N3350[23]:

```
// Accepts a const Range object, but mutates its elements
template<Iterable Rng>
void some_function(Rng const & rng) {
    ++*rng.begin(); // Deeply questionable behavior
}
```

```
template<typename I>
struct range : pair<I, I> {
    using pair<I, I>::pair;
    I begin() const { return this->first; }
    I end() const { return this->second; }
};

int main() {
    int arr[] = {1,2,3,4};
    range<int*> rng{ arr, arr + 3 };
    some_function(rng); // arr is now {2,2,3,4}
}
```

Keep in mind that the Iterable concept describes types like `vector` where a top-level const *does* affect the mutability of the iterable's elements, as well as simple pair-of-iterator style ranges like `range` above, where the top-level constness does *not* affect the elements' mutabiliy. Concepts Lite doesn't help, either. Although it's not hard to write a concept that only matches immutable iterables and use it to constrain an algorithm, it doesn't solve the problem. See below:

```
template<Iterable Rng>
concept ConstIterable =
    is_const<remove_reference_t<
        decltype(*begin(declval<Rng &>()))
    >>::value;

// This is over-constrained:
template<ConstIterable Rng>
void non_mutating_algorithm(Rng const & rng);

int rgi[] = {1,2,3,4};
range<int *> rng{rgi, rgi+4}; // a pair of mutable iterators
non_mutating_algorithm(rng); // ⟸ ERROR rng is not a ConstIterable
```

The trouble is that the type `Rng` is deduced as `range<int *>` which doesn't satisfy the requirements of the ConstIterable concept. The const in the function signature doesn't help because it doesn't change the const-ness of the iterator's reference type. Even if `rng` is const, `rng.begin()` returns an `int*`. There is no way to write the function signature such that template type deduction works *and* const-ness is applied deeply to the range's elements. That's because such a transformation would necessarily involve a change to the range's type, and type deduction makes no allowance for arbitrary type transformations during the deduction process.

3.3.13.1 But it works for `array_view`. Why?

N3851 proposes an `array_view` class for the second Library Fundamentals TS. That type allows users to independently specify the mutability of the elements and of the view with the `const` keyword. From N3851:

> A view can be created over an arbitrary *value_type*, as long as there exists a conversion from the pointer to the underlying collection object type to the pointer to the *value_type*. This allows to distinguish two levels of constness, analogously to the pointer semantics - the constness of the view and the constness of the data over which the view is defined - see Table 2. Interfaces of both the *array_view* and the *strided_array_view* allow for implicit conversions from non-const-qualified views to const-qualified views.

With this scheme, you presumably would be able to pass any Container (defined in N3851 to be arrays and class types with `size` and `data` member functions) to an algorithm as follows:

```
void some_algorithm( array_view<const int, 1> const & arr );

vector<int> vi {1,2,3,4};
array_view<int, 1> arr { vi }
some_algorithm( arr ); // OK!
```

In this case, the signature of the algorithm, together with the constructors of `array_view` and the implicit conversion of `int*` to `const int*`, conspire to make `some_function` guarantee deep-constness.

If it can work for `array_view`, why can it not work in general for all Iterables? In other words, is there no range design in which users can put `const` somewhere in their function signatures to make their algorithms deeply const? The answer, sadly, is no. Even a trivial change to the above signature shows how fragile the `array_view` design is, and why it cannot be generalized to all range types:

```
template<typename T>
void some_algorithm( array_view<const T, 1> const & arr );

vector<int> rgi{1,2,3,4};
array_view<int, 1> arr { vi }
some_algorithm( arr ); // ERROR!
```

The compiler cannot figure out what type to pick for `T` during type deduction, and there's no way for the author of `array_view` to give the compiler that guidance.

When generalized to arbitrary Iterables and Ranges, the problem is further compounded. There is no place to put a second, "deep" `const` in a signature like below:

```
template<Iterable Rng>
void some_algorithm( Rng const & rng );
```

And even if there were, the problem would be the same as for `array_view`: template type deduction would fail.

3.3.13.2 Constraints and contracts

There is no clever way around the problem in today's language, or even with Concepts Lite. This is an artifact of the style of generic programming used in C++: stating constraints on template arguments is significantly different than adding `const`-ness to a concrete type or template specialization.

A template's constraints define part of that template's contract, and this contract applies not only to the user of that template, but also its implementer. In the earlier example, `some_function` requires its template arguments to satisfy the requirements of the Iterable concept. Under that contract, the implementation should use only the expressions allowed by the concept on objects whose types are constrained by the concept. In other words, those requirements define a set of expressions and types that are *admissible* within the template.

Recall the `some_function` define above:

```
// Accepts a const Range object, but mutates its elements
template<Iterable Rng>
void some_function(Rng const & rng) {
    ++*rng.begin(); // Deeply questionable behavior
}
```

The expression `++*rng.begin()` invokes not one, but two operations not admitted by the requirements of the Iterable concept:

- `rng.begin()` (which should be `using std::begin; begin(rng)`), and
- `++x` where `x` is the object returned by `*begin(rng)`.

While this program is syntactically well-formed, and instantiation may even succeed, its behavior would be undefined under the terms of its contract. In other words, the implementation does not conform to its constraints. This is not fundamentally different than using a `const_cast` to modify the value of a `const`-qualified function argument: it might work, but the modification invokes undefined behavior.

Every proposed range design will have the same problem. The lack of a library solution to this problem should not hold up a proposal. A work-around is for users to test each algorithm with an archetypal non-mutable iterator, as has been done with the Boost Concept Check Library[3]. The standard library could make this task simpler by providing debug archetypal iterators for testing purposes.

3.3.14 Singular Ranges Are Not Empty

Ranges are default-constructible. The question is, what can be done with a such a range? Given a Range type `R` and an object `r` defined as follows:

```
R r;                            // "singular" range
assert( begin(r) == end(r) );   // Is this OK?
```

Default-constructed standard containers are empty. Can we expect the same is true for ranges? No, we cannot. The intention is that a simple pair of iterators may qualify as a valid range. The section [iterator.requirements.general]/5 explains what can be done with singular iterators, and comparing them for equality is not listed as one of the valid operations. Therefore, singular ranges are not “empty”. In terms of EoP, they are partially formed objects that must be assigned to before they can be used 8.

A particular model of the Range concept may choose to provide this guarantee, but it is not required by the Range concept itself and, as such, should not be relied upon.

4 Concept Definitions

The following concepts are proposed to constrain the standard library. The iterator concepts mentioned here are identical to those specified in N3351[21] except where specified.

4.1 Iterator Concepts

The range concepts presented below build on the following iterator concepts. These are largely as found in N3351[21], with the addition of WeakIterator, Iterator, WeakOutputIterator and OutputIterator. The entire hierarchy is presented here for completeness. An important, new concept is:

```
concept WeakIterator<typename I> =
    WeaklyIncrementable<I> && Copyable<I> &&
    requires(I i) {
        { *i } → auto&&;
    };
```

A WeakIterator is a WeaklyIncrementable type that is both copyable and dereferencable, in addition to being pre-incrementable. The only requirement on the result of dereferencing the iterator is that its type can be deduced from `auto&&`. Effectively, the result of dereferencing shall not be `void`. The WeakIterator concept allows iterator adaptors to provide a dereferencing operator for both InputIterators and OutputIterators.

This concept is refined by the addition of either Readable or Writable, which allow reading from and writing to a dereferenced iterator, respectively. The Iterator and WeakIterator also obviate the need for separate InputRange and OutputRange concepts (as we see later).

```
concept Iterator<typename I> =
    WeakIterator<I> && EqualityComparable<I>;

concept WeakOutputIterator<typename I, typename T> =
    WeakIterator<I> && Writable<I, T>;
```

```
concept OutputIterator<typename I, typename T> =
    WeakOutputIterator<I> && EqualityComparable<I>;
```

N3351 does not define WeakOutputIterator or OutputIterator, preferring instead to define all the algorithms in terms of WeaklyIncrementable and (separately) Writable. We see no disadvantage to rolling these two into the familiar (Weak)OutputIterator concept as a convenience; it saves typing.

The WeakInputIterator concept defines requirements for a type whose referred to values can be read (from the requirement for Readable) and which be both pre- and post-incremented. However, WeakInputIterators are not required to be compared for equality. There are a number of algorithms whose input ranges are defined by the equality of comparison of another input range (e.g., `std::mismatch` and `std::equal`).

```
concept WeakInputIterator<WeakIterator I> =
    Readable<I> &&
    requires(I i) {
        typename IteratorCategory<I>;
        { i++ } → Readable;
        requires Derived<IteratorCategory<I>, weak_input_iterator_tag>;
    };
```

An InputItreator is a WeakInputIterator that can be compared for equality comparison. This concept defines the basic requirements for a very large number of the algorithm in the standard library.

```
concept InputIterator<typename I> =
    WeakInputIterator<I> &&
    EqualityComparable<I> &&
    Derived<IteratorCategory<I>, input_iterator_tag>;
```

The ForwardIterator concept refines the InputIterator concept, but the refinement is *semantic*. It guarantees the “multipass” property: it allows multiple iterations of a range of elements. This is something that cannot be done with, say, an `istream_iterator`. The current element is consumed when the iterator is incremented.

```
concept ForwardIterator<typename I> =
    InputIterator<I> &&
    Incrementable<I> &&
    Derived<IteratorCategory<I>, forward_iterator_tag>;
```

The BidirectionalIterator concept refines the ForwardIterator concept, and allows both incrementing and decrementing.

```
concept BidirectionalIterator<typename I> =
    ForwardIterator<I> &&
    Derived<IteratorCategory<I>, bidirectional_iterator_tag> &&
    requires decrement (I i, I j) {
        { --i } → I&;
        { i-- } → I;
    };
```

The RandomAccess concept refines the BidirectionalIterator concept and provides support for constant-time advancement using `+=`, `+`, and `-=`, and the computation of distance in constant time using `-`. Random access iterators also support array notation via subscripting.

```
concept RandomAccessIterator<typename I> =
    BidirectionalIterator<I> &&
    TotallyOrdered<I> &&
    Derived<IteratorCategory<I>, random_access_iterator_tag> &&
    SignedIntegral<DistanceType<I>> &&
    SizedIteratorRange<I, I> && // see below
    requires advance (I i, I j, DifferenceType<I> n) {
        { i += n } → I&;
        { i + n } → I;
        { n + i } → I;
        { i -= n } → I&;
        { i - n } → I;
        { i[n] } → ValueType<I>;
    };
```

4.2 Iterator Range Concepts

The IteratorRange concept defines a pair of types (an Iterator and a Sentinel), that can be compared for equality. This concept is the key that allows iterator ranges to be defined by pairs of types that are not the same.

```
concept IteratorRange<typename I, typename S> =
    Iterator<I> &&
    Regular<S> &&
    EqualityComparable<I, S>;
```

The SizedIteratorRange allows the use of the - operator to compute the distance between to an Iterator and a Sentinel.

```
concept SizedIteratorRange<typename I, typename S> =
    IteratorRange<I, S> &&
    requires difference (I i, S j) {
        typename Distance_type<I>;
        { i - i } → Distance_type<I>;
        { j - j } → Distance_type<I>;
        { i - j } → Distance_type<I>;
        { j - i } → Distance_type<I>;
        requires SignedIntegral<DifferenceType>;
    };
```

4.3 Iterable Concepts

This proposal introduces a new concept for describing the requirements on algorithms that require access to the `begin` and `end` of a Range or Container.

```
concept Iterable<typename T> =
    requires(T t) {
        typename IteratorType<T>;
```

```
        typename SentinelType<T>;
        { begin(t) } -> IteratorType<T>;
        { end(t) } -> SentinelType<T>;
        requires IteratorRange<IteratorType, SentinelType>;
    }
```

The Iterable concept requires two associated operations: `begin`, and `end`. Note that the return types of these operations are not required to be the same. The names of those return types are `IteratorType<T>` and `SentinelType<T>`, respectively. Furthermore, that pair of types must satisfy the requirements of the IteratorRange concept defined in the previous section.

Iterable types have no other requirements. Most algorithms requiring this concept simply forward to an Iterator-based algorithm by calling `begin` and `end`.

The Range concept refines the Iterable concept and defines a view on a range of elements that it does not own.

```
concept Range<typename T> =
    Iterable<T> && Semiregular<T> && is_range<T>::value;
```

A possible implementation of the `is_range` predicate is shown below.

```
// For exposition only:
struct range_base
{};

concept ContainerLike<Iterable T> =
    !Same<decltype(*begin(declval<T &>())),
          decltype(*begin(declval<T const &>()))>;

// For exposition only:
template<typename T>
struct is_range_impl_
  : std::integral_constant<
        bool,
        Iterable<T> && (!ContainerLike<T> || Derived<T, range_base>)
    >
{};

// Specialize this if the default is wrong.
template<typename T, typename Enable = void>
struct is_range
  : conditional<
        is_same<T, remove_const_t<remove_reference_t<T>>>::value,
        is_range_impl_<T>,
        is_range<remove_const_t<remove_reference_t<T>>>
    >::type
{};
```

Note that the “multipass” property–the property that guarantees that you can iterate the same range twice, accessing the same objects in each pass–is guaranteed by the `forward_iterator_tag` class. That is, any iterator having whose

iterator category is derived from that tag class is required to be a multipass iterator.

4.4 Sized Iterable Concepts

The `SizedIterable` concept exists for the same reasons as `SizedIteratorRange`. There are some iterables that, though they are not random-access, know their size in O(1). A prime example is `std::list`. Another example is a counted view (i.e., a range specified by an iterator and a count). Some algorithms can select a better implementation when the size of the range is known, even if the iterators don't allow random access. (One example is a `search` algorithm that knows to stop searching when there is no longer room in the input sequence for the pattern to match.)

```
// For exposition only:
concept SizedIterableLike<Iterable R> =
    requires(R r) {
        typename SizeType<R>;
        { size(r) } → SizeType<R>; // ADL customization point
        requires Integral<SizeType>;
    }

concept SizedIterable<SizedIterableLike R> =
    is_sized_iterable<R>::value;
```

Like N4017[10], we propose `size(r)` as a customization point for accessing the size of an Iterable. A SizedIterable requires that `size(r)` returns the same value as `distance(begin(r), end(r))` (module a possible change in the signed-ness of the Integral type). As in N4017, we suggest `size` overloads in `std::` for arrays and for class types with a `size()` member.

In recognition of the fact that nominal conformance to a concept is insufficiently fine-grained, the SizedIterable concept is defined in terms of a `is_sized_iterable` trait. This is to give users explicit control in the case of accidental conformance with the syntactic requirements of the concept. A possible default implementation of the `is_sized_iterable` trait is shown below.

```
// For exposition only:
template<typename R>
struct is_sized_iterable_impl_
  : std::integral_constant<
        bool,
        SizedIterableLike<R>
    >
{};

// Specialize this if the default is wrong.
template<typename R>
struct is_sized_iterable
  : conditional<
        is_same<R, remove_const_t<remove_reference_t<R>>>::value,
        is_sized_iterable_impl_<R>,
```

```
        is_sized_iterable<remove_const_t<remove_reference_t<R>>>
    >::type
{};
```

5 Technical Specifications

This section is intentionally left blank.

6 Future Directions

More work is necessary to get ranges into the standard. Subsequent proposals will recommend specific components for standardization as described in Impact on the Standard. In addition, below are some additional areas that are currently open for research.

- Range extensions to things like regex. Make it work with null-terminated strings, e.g..
- Discuss the pros and cons of tying this work with work on a concept-ified standard library (aka Concepts Lite TS2).
- Discuss the pros and cons of making this (and the concept-ified standard library) *strictly* backwards compatible, versus delivering the new, conceptified and range-ified standard library in a separate, versioned namespace (and what such a solution should look like).
- Discuss how an extension to the Concepts Lite proposal, implicit conversion to concept, bears on the Iterable/Range concepts and the way the algorithms are constrained. Also discuss how it can give a syntax to express “this function does not mutate the range elements.”
- Assess the impact of allowing ForwardIterators to return proxies.
- Consider integrating ContiguousIterator from N4132[11]

7 Acknowledgements

I would like to give special thanks to Sean Parent for his advice and feedback on early designs of the range library on which this proposal is based, in addition to his work on the Adobe Source Libraries[1] from which this proposal has borrowed liberally.

Also deserving of special thanks is Andrew Sutton. His work on Concepts Lite and on the formulations of the algorithms as specified in N3351[21] has proven invaluable, and he has generously donated his time and expertise to expound on the ideas there and improve the quality of this proposal.

Chandler Carruth has also helped more than he probably knows. I am indebted to him for his support and perspective.

I would be remiss if I didn’t acknowledge the foundational work of all the people whose ideas and sweat have gone into various range libraries and proposals in the past. They are too many to list, but I certainly benefited from the work of Dave Abrahams, Dietmar Kühl, Neil Groves, Thorsten Ottosen, Arno Schoedl, Daniel Walker, and Jeremy Seik. Neil Groves also submitted a particularly extensive review of this document.

Of course none of this work would be possible without Alex Stepanov’s giant leap forward with the STL, or without Bjarne Stroustrup who gave Alex the instrument he needed to most clearly realize his vision.

8 References

[1] Adobe Source Libraries: *http://stlab.adobe.com*. Accessed: 2014-10-08.

[2] Austern, M. 2000. Segmented Iterators and Hierarchical Algorithms. *Selected Papers from the International Seminar on Generic Programming* (2000), 80–90.

[3] Boost Concept Check Library: *http://boost.org/libs/concept_check*. Accessed: 2014-10-08.

[4] Boost.Range Library: *http://boost.org/libs/range*. Accessed: 2014-10-08.

[5] Clang Modernize: *http://clang.llvm.org/extra/clang-modernize.html*. Accessed: 2014-10-08.

[6] D Phobos std.range: *http://dlang.org/phobos/std_range.html*. Accessed: 2014-10-08.

[7] Debug info: Support fragmented variables: *http://reviews.llvm.org/D2680*. Accessed: 2014-10-08.

[8] Dietmar, K. and Abrahams, D. 2005. N1873: The Cursor/Property Map Abstraction.

[9] libc++ C++ Standard Library: *http://libcxx.llvm.org/*. Accessed: 2014-10-08.

[10] Marcangelo, R. 2014. N4017: Non-member size() and more.

[11] Maurer, J. 2014. N4132: Contiguous Iterators.

[12] Muchnick, S. 1997. Advanced Compiler Design Implementation. Morgan Kaufmann.

[13] New Iterator Concepts: *http://www.boost.org/libs/iterator/doc/new-iter-concepts.html*. Accessed: 2014-10-08.

[14] NTCTS Iterator: https://github.com/Beman/ntcts_iterator. Accessed: 2014-10-08.

[15] Parent, S. 2013. C++ Seasoning. *GoingNative 2013*.

[16] Position-Based Ranges: https://github.com/Bekenn/range. Accessed: 2014-10-08.

[17] Range Comprehensions: http://ericniebler.com/2014/04/27/range-comprehensions/. Accessed: 2014-10-08.

[18] Range v3: http://www.github.com/ericniebler/range-v3. Accessed: 2014-10-08.

[19] Schödl, A. and Fracassi, F. 2013. N3782: Index-Based Ranges.

[20] Stepanov, A. and McJones, P. 2009. *Elements of Programming*. Addison-Wesley Professional.

[21] Stroustrup, B. and Sutton, A. 2012. N3351: A Concept Design for the STL.

[22] Universal References in C++11: http://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers. Accessed: 2014-10-08.

[23] Yasskin, J. 2012. N3350: A minimal std::range.

9 Appendix 1: Sentinels and Code Generation

In this appendix we explore the effect of sentinels on code generation. I'll show that allowing the type of the end iterator to differ from the begin can have a positive effect on the performance of algorithms. First, I'll note that nothing that can be done with sentinels cannot also be done with appropriately designed end iterators. Here, for instance, is the code for an iterator that can be used to adapt a null-terminated string to the STL. It is implemented with the help of the Boost.Iterators library:

```
#include <cassert>
#include <iostream>
#include <boost/iterator/iterator_facade.hpp>

struct c_string_range
{
private:
    char const *str_;
public:
    using const_iterator = struct iterator
      : boost::iterator_facade<
            iterator
          , char const
```

```
          , std::forward_iterator_tag
        >
    {
    private:
        friend class boost::iterator_core_access;
        friend struct c_string_range;
        char const * str_;
        iterator(char const * str)
          : str_(str)
        {}
        bool equal(iterator that) const
        {
            return str_
                ? (that.str_ == str_ ||
                     (!that.str_ && !*str_))
                : (!that.str_ || !*that.str_);
        }
        void increment()
        {
            assert(str_ && *str_);
            ++str_;
        }
        char const& dereference() const
        {
            assert(str_ && *str_);
            return *str_;
        }
    public:
        iterator()
          : str_(nullptr)
        {}
    };
    c_string_range(char const * str)
      : str_(str)
    {
        assert(str_);
    }
    iterator begin() const
    {
        return iterator{str_};
    }
    iterator end() const
    {
        return iterator{};
    }
    explicit operator bool() const
    {
        return !!*str_;
    }
};

int c_strlen(char const *sz)
{
    int i = 0;
    for(; *sz; ++sz)
        ++i;
```

```
    return i;
}

int range_strlen(
    c_string_range::iterator begin,
    c_string_range::iterator end)
{
    int i = 0;
    for(; begin ≠ end; ++begin)
        ++i;
    return i;
}
```

The code traverses the sequence of characters without first computing its end. It does it by creating a dummy end iterator such that any time a real iterator is compared to it, it checks to see if the real iterator points to the null terminator. All the comparison logic is in the `c_string_range::iterator::equal` member function.

The functions `c_strlen` and `range_strlen` implement equivalent procedures for computing the length of a string, the first using raw pointers and a check for the null terminator, the second using `c_string_range`'s STL iterators. The resulting optimized assembly code (clang 3.4 -O3 -DNDEBUG) generated for the two functions highlights the lost optimization opportunities.

`c_strlen`

```
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %ecx
    xorl    %eax, %eax
    cmpb    $0, (%ecx)
    je      LBB1_3
    xorl    %eax, %eax
    .align  16, 0x90
LBB1_2:
    cmpb    $0, 1(%ecx,%eax)
    leal    1(%eax), %eax
    jne     LBB1_2
LBB1_3:
    popl    %ebp
    ret
```

`range_strlen`

```
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %esi
    leal    8(%ebp), %ecx
    movl    12(%ebp), %esi
    xorl    %eax, %eax
    testl   %esi, %esi
    movl    8(%ebp), %edx
    jne     LBB2_4
    jmp     LBB2_1
    .align  16, 0x90
LBB2_8:
    incl    %eax
    incl    %edx
    movl    %edx, (%ecx)
LBB2_4:
    testl   %edx, %edx
    jne     LBB2_5
    cmpb    $0, (%esi)
    jne     LBB2_8
    jmp     LBB2_6
    .align  16, 0x90
LBB2_5:
    cmpl    %edx, %esi
    jne     LBB2_8
    jmp     LBB2_6
    .align  16, 0x90
```

```
LBB2_3:
    leal    1(%edx,%eax), %esi
    incl    %eax
    movl    %esi, (%ecx)
LBB2_1:
    movl    %edx, %esi
    addl    %eax, %esi
    je      LBB2_6
    cmpb    $0, (%esi)
    jne     LBB2_3
LBB2_6:
    popl    %esi
    popl    %ebp
    ret
```

Code like `c_string_range` exists in the wild; for instance, see Beman Dawes' ntcts_iterator[14]. But more typically, when users want to use an STL algorithm on a C-style string, they call `strlen` to find the end first (this is what the standard regex algorithms do when passed C-style strings). That traverses the string an extra time needlessly. Also, such a trick is not possible for input sequences like those traversed by `std::istream_iterator` that consume their input.

Rather than mocking up a dummy end iterator with a computationally expensive equality comparison operation, we can use a sentinel type that encodes end-ness in its type. Below is an example from the Range-v3 library[18], which uses a `range_facade` class template to generate iterators and sentinels from a simple range-like interface:

```
using namespace ranges;
struct c_string_iterable
  : range_facade<c_string_iterable>
{
private:
    friend range_access;
    char const *sz_;
    char const & current() const { return *sz_; }
    void next() { ++sz_; }
    bool done() const { return *sz_ == 0; }
    bool equal(c_string_iterable const &that) const
    { return sz_ == that.sz_; }
public:
    c_string_iterable() = default;
    c_string_iterable(char const *sz)
        : sz_(sz) {}
};

// Iterable-based
int iterable_strlen(
    range_iterator_t<c_string_iterable> begin,
    range_sentinel_t<c_string_iterable> end)
{
    int i = 0;
    for(; begin != end; ++begin)
```

```
        ++i;
    return i;
}
```

The assembly generated for `iterable_strlen` is nearly identical to that for the hand-coded `c_strlen`:

```
c_strlen                        iterable_strlen

    pushl   %ebp                    pushl   %ebp
    movl    %esp, %ebp              movl    %esp, %ebp
    movl    8(%ebp), %ecx           movl    8(%ebp), %ecx
    xorl    %eax, %eax              xorl    %eax, %eax
    cmpb    $0, (%ecx)              cmpb    $0, (%ecx)
    je      LBB1_3                  je      LBB1_4
    xorl    %eax, %eax              leal    8(%ebp), %edx
    .align  16, 0x90                .align  16, 0x90
LBB1_2:                         LBB1_2:
    cmpb    $0, 1(%ecx,%eax)        cmpb    $0, 1(%ecx,%eax)
    leal    1(%eax), %eax           leal    1(%eax), %eax
    jne     LBB1_2                  jne     LBB1_2
                                    addl    %eax, %ecx
                                    movl    %ecx, (%edx)
LBB1_3:                         LBB1_4:
    popl    %ebp                    popl    %ebp
    ret                             ret
```

The generated code for `iterable_strlen` is better than for `range_strlen` because the sentinel has a different type than the iterator, so that the expression `begin` ≠ `end` can be optimized into `*begin` = `0`. In `range_strlen`, `begin` = `end` is comparing two objects of the same type, and since either `begin` or `end` could be a sentinel - or both, or neither - the compiler can't elide the extra checks without extra information from the surrounding calling context, which isn't always available; hence, the worse code gen.

In addition to the performance impact, the complexity of implementing a correct `operator`= for an iterator with a dummy sentinel can present problems. Chandler Carruth reports that such comparison operators have been a rich source of bugs for Google.

10 Appendix 2: Sentinels, Iterators, and the Cross-Type EqualityComparable Concept

This appendix describes the theoretical justification for sentinels from the perspective of the STL concepts as set out in N3351[21]. In that paper, the foundational concept EqualityComparable is described in depth, not only its

syntactic constraints but also its semantic axioms. It is not enough that the syntax `a == b` compiles. It has to be a meaningful comparison. Here I explain why I believe it is meaningful to compare an iterator with a sentinel of a different type for equality.

In the expression `x == y`, where `x` and `y` have different types, the EqualityComparable concept requires that the types of both `x` and `y` must themselves be EqualityComparable, and there must be a common type to which they can both be converted, and that type must also be EqualityComparable. Think of comparing a `char` with a `short`. It works because both `char` and `short` are EqualityComparable, and because they can both be converted to an `int` which is also EqualityComparable.

Iterators are comparable, and sentinels are trivially comparable (they always compare equal). The tricky part is the common type requirement. Logically, every iterator/sentinel pair has a common type that can be constructed as follows: assume the existence of a new iterator type `I` that is a tagged union that contains *either* an iterator *or* a sentinel. When an iterator is compared to a sentinel, it behaves semantically as if both the iterator and the sentinel were first converted to two objects of type `I` — call them `lhs` and `rhs` — and then compared according to the following truth table:

LHS IS SENTINEL ?	RHS IS SENTINEL ?	LHS == RHS ?
true	true	true
true	false	done(rhs.iter)
false	true	done(lhs.iter)
false	false	lhs.iter == rhs.iter

[*]

In Appendix 1, there is an implementation of `c_string_range` whose iterator's `operator==` is a procedure for evaluating this truth table. That's no coincidence; that was a special case of this more general construction.

In summary, for every iterator/sentinel pair, we can construct a common iterator type that implements an equivalent procedure for computing equality. The existence of this common type is what allows iterator/sentinel pairs to satisfy the EqualityComparable requirement.

As a final note, the Range v3 library has a general implementation of this common iterator type as a parametrized type, and appropriate specializations of `std::common_type` that allow the constrained algorithms to type-check correctly. It works well in practice, both for the purpose of type-checking the algorithms and for adapting ranges with iterator/sentinel pairs to old code that expects the begin and end of a range to have the same type.

10.1 Sentinel Equality

The first line of table above shows that sentinels always compare equal - regardless of their state. This can seem unintuitive in some situations. For instance:

```
auto s = "Hello World!?";
// Hypothetical range-like facilities for exposition only:
auto r1 = make_range( &s[0], equal_to('!') );
auto r2 = make_range( &s[0], equal_to('?') );
// 'end()' returns each 'equal_to' sentinel object:
assert(r1.end() == r2.end() && "really?");
```

In the above example, it's clear that `r1.end()` and `r2.end()` are refering to different elements, and that's reflected in their having different state. So shouldn't sentinel equality be a stateful comparison?

It's incorrect to think of sentinels as objects whose position is determined by their value. Thinking that way leads to logical inconsistencies. It's not hard to define sentinels that - if you consider their state during comparison - compare equal when they represent a different position, and not equal when they represent the same position. Consider:

```
auto s = "hello! world!";
auto r1 = make_range( &s[0], equal_to('!') );
auto r2 = make_range( &s[7], equal_to('!') );

assert(r1.end() == r2.end() && "really?");
```

In the above code, although the sentinels have the same type and the same state, they refer to different elements.

Also imagine a counted iterator that stores an underlying iterator and a count that starts at zero. It's paired with a sentinel that contains an ending count:

```
auto s = "hello";
// Hypothetical range-like facilities for exposition only:
auto r1 = make_range( counting(&s[0]), end_count(5) );
auto r2 = make_range( counting(&s[1]), end_count(4) );

assert(r1.end() ≠ r2.end() && "really?");
```

The end sentinels have the same type and different state, but they refer to the same element.

The question then is: What does it *mean* to compare sentinels for equality? The meaning has to be taken from the context in which the operation appears. In that context - the algorithms - the sentinel is one of two positions denoting a range.

Iterator comparison is a position comparison. A sentinel, when it's paired with an iterator as in the algorithms, denotes a unique position. Since iterators must be valid sentinels, comparing sentinels should also be a position comparison. In this context, sentinels always represent the same

position – the end – even though that position is not yet know. Hence, sentinels compare equal always, without any need to consider state. Anything else yields results that are inconsistent with the use of iterators to denote the end of a range.

What about the examples above where sentinel comparison seems to give nonsensical results? In those examples, we are expecting sentinels to have meaning outside of their valid domain – when they are considered in isolation of their paired iterator. It is the paired iterator that give the sentinel its “unique-position-ness”, so we shouldn’t expect an operation that compares position to make sense on an object that is not capable of representing a position by itself.

11 Appendix 3: D Ranges and Algorithmic Complexity

As currently defined, D’s ranges make it difficult to implement algorithms over bidirectional sequences that take a position in the middle as an argument. A good example is a hypothetical `is_word_boundary` algorithm. In C++ with iterators, it might look like this:

```
template< BidirectionalIterator I >
void is_word_boundary( I begin, I middle, I end )
{
    bool is_word_prev = middle == begin ? false : isword(*prev(middle));
    bool is_word_this = middle == end ? false : isword(*middle);
    return is_word_prev != is_word_this;
}
```

Users might call this in a loop to find the first word boundary in a range of characters as follows:

```
auto i = myrange.begin();
for( ; i != myrange.end(); ++i )
    if( is_word_boundary( myrange.begin(), i, myrange.end() ) )
        break;
```

The question is how such an API should be designed in D, since D doesn’t have iterators. In a private email exchange, Andrei Alexandrescu, the designer of D’s range library, described this potential implementation (the detailed implementation is ours):

```
bool is_word_boundary(Range1, Range2)( Range1 front, Range2 back )
    if (isBidirectionalRange!Range1 && isInputRange!Range2 )
{
    bool is_word_prev = front.empty ? false : isword(front.back);
    bool is_word_this = back.empty ? false : isword(back.front);
    return is_word_prev != is_word_this;
}
```

```
range r = myrange;
size_t n = 0;
for(range r = myrange; !r.empty; r.popFront(), ++n)
     if( is_word_boundary( takeExactly(myrange, n), r) )
        break;
```

This example uses D's `takeExactly` range adaptor. `takeExactly` is like Haskell's `take` function which creates a list from another list by taking the first `n` elements, but `takeExactly` requires that the input range has at least `n` elements begin with.

The trouble with the above implementation is that `takeExactly` demotes Bidirectional ranges to Forward ranges. For eample, taking the first *N* elements of a linked list cannot give access to the final element in O(1). So the `for` loop is trying to pass a Forward range to an algorithm that clearly requires a Bidirectional range. The only fix is to loosen the requirements of `Range1` to be Forward. To do that, the implementation of `is_word_boundary` needs to change so that, when it is passed a Forward range, it walks to the end of the `front` range and tests the last character. Obviously, that's an O(N) operation.

In other words, by converting the `is_word_boundary` from iterators to D-style ranges, the algorithm goes from O(1) to O(N). From this we draw the following conclusion: D's choice of algorithmic *basis operations* is inherently less efficient than C++'s.

12 Appendix 4: On Counted Ranges and Efficiency

The three types of ranges that we would like the Iterable concept to be able to efficiently model are:

1. Two iterators
2. An iterator and a predicate
3. An iterator and a count

The Iterator/Sentinel abstraction is what makes it possible for the algorithms to handle these three cases with uniform syntax. However, the third option presents challenges when trying to make some algorithms optimally efficient.

Counted ranges, formed by specifying a position and a count of elements, have iterators – as all Iterables do. The iterators of a counted range must know the range's extent and how close they are to reaching it. Therefore, the counted range's iterators must store both an iterator into the underlying sequence and a count – either a count to the end or a count from the front. Here is one potential design:

```
class counted_sentinel
{};

template<WeakIterator I>
class counted_iterator
{
    I it_;
    DistanceType<I> n_; // distance to end
public:
    // ... constructors...
    using iterator_category =
        typename iterator_traits<I>::iterator_category;
    decltype(auto) operator*() const
    {
        return *it_;
    }
    counted_iterator & operator++()
    {
        ++it_;
        --n_;
        return *this;
    }
    friend bool operator==(counted_iterator const & it,
                           counted_sentinel)
    {
        return it.n_ == 0;
    }
    // ... other operators...
};

template<WeakIterator I>
class counted_range
{
    I begin_;
    DistanceType<I> count_;
public:
    // ... constructors ...
    counted_iterator<I> begin() const
    {
        return {begin_, count_};
    }
    counted_sentinel end() const
    {
        return {};
    }
};

template<WeakIterator I>
struct common_type<counted_iterator<I>, counted_sentinel>
    // ... see Appendix 2 ...
```

There are some noteworthy things about the code above. First, `counted_iterator` bundles an iterator and a count. Right off, we see that copying counted iterators is going to be more expensive, and iterators are copied frequently. A mitigating factor is that the sentinel is empty. Passing a `counted_iterator` and a `counted_sentinel` to an algorithm copies as much data as passing an

iterator and a count. When passed separately, the compiler probably has an easier time fitting them in registers, but some modern compilers are capable passing the members of a struct in registers. This compiler optimization is sometimes called Scalar Replacement of Aggregates (see Muchnick 1997, ch. 12.2 [12]) and is known to be implemented in gcc and LLVM (see this recent LLVM commit[7] for example).

Also, incrementing a counted iterator is expensive: it involves incrementing the underlying iterator and decrementing the internal count. To see why this is potentially expensive, consider the trivial case of passing a `counted_iterator<list<int>::iterator>` to `advance`. That counted iterator type is bidirectional, and `advance` must increment it *n* times:

```
template<BidirectionalIterator I>
void advance(I & i, DistanceType<I> n)
{
    if(n ⩾ 0)
        for(; n ≠ 0; --n)
            ++i;
    else
        for(; n ≠ 0; ++n)
            --i;
}
```

Notice that for each `++i` or `--i` here, *two* increments or decrements are happening when `I` is a `counted_iterator`. This is sub-optimal. A better implementation for `counted_iterator` is:

```
template<BidirectionalIterator I>
void advance(counted_iterator<I> & i, DistanceType<I> n)
{
    i.n_ -= n;
    advance(i.it_, n);
}
```

This has a noticeable effect on the generated code. As it turns out, `advance` is one of the relatively few places in the standard library where special handling of `counted_iterator` is advantageous. Let's examine some algorithms to see why that's the case.

12.1 Single-Pass Algorithms with Counted Iterators

First, let's look at a simple algorithm like `for_each` that makes exactly one pass through its input sequence:

```
template<InputIterator I, Regular S, Function<ValueType<I>> F>
    requires EqualityComparable<I, S>
I for_each(I first, S last, F f)
{
    for(; first ≠ last; ++first)
        f(*first);
```

```
    return first;
}
```

When passed counted iterators, at each iteration of the loop, we do an increment, a decrement (for the underlying iterator and the count), and a comparison. Let's compare this against a hypothetical `for_each_n` algorithm that takes the underlying iterator and the count separately. It might look like this:

```
template<InputIterator I, Function<ValueType<I>> F>
I for_each_n(I first, DifferenceType<I> n, F f)
{
    for(; n ≠ 0; ++first, --n)
        f(*first);
    return first;
}
```

For the hypothetical `for_each_n`, at each loop iteration, we do an increment, a decrement, and a comparison. That's exactly as many operations as `for_each` does when passed counted iterators. So a separate `for_each_n` algorithm is probably unnecessary if we have sentinels and `counted_iterator`s. This is true for any algorithm that makes only one pass through the input range. That turns out to be a lot of algorithms.

12.2 Multi-Pass Algorithms with Counted Iterators

There are other algorithms that make more than one pass over the input sequence. Most of those, however, use `advance` when they need to move iterators by more than one hop. Once we have specialized `advance` for `counted_iterator`, those algorithms that use `advance` get faster without any extra work.

Consider `partition_point`. Here is one example implementation, taken from libc++ [9] and ported to Concepts Lite and sentinels:

```
template<ForwardIterator I, Regular S, Predicate<ValueType<I>> P>
    requires EqualityComparable<I, S>
I partition_point(I first, S last, P pred)
{
    DifferenceType<I> len = distance(first, last);
    while (len ≠ 0)
    {
        DifferenceType<I> l2 = len / 2;
        I m = first;
        advance(m, l2);
        if (pred(*m))
        {
            first = ++m;
            len -= l2 + 1;
        }
        else
            len = l2;
    }
```

```
    return first;
}
```

Imagine that `I` is a forward `counted_iterator` and `S` is a `counted_sentinel`. If the library does not specialize `advance`, this is certainly inefficient. Every time `advance` is called, needless work is being done. Compare it to a hypothetical `partition_point_n`:

```
template<ForwardIterator I, Predicate<ValueType<I>> P>
I partition_point_n(I first, DifferenceType<I> len, P pred)
{
    while (len ≠ 0)
    {
        DifferenceType<I> l2 = len / 2;
        I m = first;
        advance(m, l2);
        if (pred(*m))
        {
            first = ++m;
            len -= l2 + 1;
        }
        else
            len = l2;
    }
    return first;
}
```

The first thing we notice is that `partition_point_n` doesn't need to call `distance`! The more subtle thing to note is that calling `partition_point_n` with a raw iterator and a count saves about O(N) integer decrements over the equivalent call to `partition_point` with `counted_iterators` … unless, of course, we have specialized the `advance` algorithm as shown above. Once we have, we trade the O(N) integer decrements for O(log N) integer subtractions. That's a big improvement.

But what about the O(N) call to `distance`? Actually, that's easy, and it's the reason why the SizedIteratorRange concept exists. `counted_iterator` stores the distance to the end. So the distance between a `counted_iterator` and a `counted_sentinel` (or between two `counted_iterators`) is known in O(1) *regardless of the iterator's category*. The SizedIteratorRange concept tests whether an iterator `I` and a sentinel `S` can be subtracted to get the distance. This concept is modeled by random-access iterators by their nature, but also by counted iterators and their sentinels. The `distance` algorithm is specialized for SizedIteratorRange, so it is O(1) for counted iterators.

With these changes, we see that `partition_point` with counted iterators is very nearly as efficient as a hypothetical `partition_point_n` would be, and we had to make no special accommodations. Why can't we make `partition_point` *exactly* as efficient as `partition_point_n`? When `partition_point` is called with a counted iterator, it also *returns* a counted iterator. Counted iterators contain two datums: the position and distance to the end. But when `partition_point_n` returns just the position, it is actually computing and returning less information.

Sometimes users don't need the extra information. But sometimes, after calling `partition_point_n`, the user might want to pass the resulting iterator to another algorithm. If *that* algorithm calls `distance` (like `partition_point` and other algorithms do), then it will be O(N). With counted iterators, however, it's O(1). So in the case of `partition_point`, counted iterators cause the algorithm to do O(log N) extra work, but it sometimes saves O(N) work later.

To see an example, imagine a trivial `insertion_sort` algorithm:

```
template<ForwardIterator I, Regular S>
    requires EqualityComparable<I, S> && Sortable<I> // from N3351
void insertion_sort(I begin, S end)
{
    for(auto it = begin; it ≠ end; ++it)
    {
        auto insertion = upper_bound(begin, it, *it);
        rotate(insertion, it, next(it));
    }
}
```

Imagine that `I` is a `counted_iterator`. The first thing `upper_bound` does is call `distance`. Making `distance` O(1) for `counted_iterator`s saves N calls of an O(N) algorithm. To get comparable performance for an equivalent procedure in today's STL, users would have to write a separate `insertion_sort_n` algorithm that dispatches to an `upper_bound_n` algorithm – that they would also need to write themselves.

12.3 Counted Algorithms with Counted Iterators

We've seen that regular algorithms with counted iterators can be made nearly as efficient as dedicated counted algorithms, and that sometimes we are more than compensated for the small performance loss. All is not roses, however. There are a number of *counted algorithms* in the standard (the algorithms whose names end with `_n`). Consider `copy_n`:

```
template<WeakInputIterator I, WeakOutputIterator<ValueType<I>> O>
pair<I, O> copy_n(I in, DifferenceType<I> n, O out)
{
    for(; n ≠ 0; ++in, ++out, --n)
        *out = *in;
    return {in, out};
}
```

(We've changed the return type of `copy_n` so as not to lose information.) If `I` is a counted iterator, then for every `++in`, an increment and a decrement are happening, and in this case the extra decrement is totally unnecessary. For *any* counted (i.e., `_n`) algorithm, something special needs to be done to keep the performance from degrading when passed counted iterators.

The algorithm author has two options here, and neither of them is ideal.

Option 1: Overload the algorithm

The following is an optimized version of `copy_n` for counted iterators:

```
template<WeakInputIterator I, WeakOutputIterator<ValueType<I>> O>
pair<I, O> copy_n(counted_iterator<I> in, DifferenceType<I> n, O out)
{
    for(auto m = in.n_ - n; in.n_ ≠ m; ++in.i_, --in.n_, ++out)
        *out = *in;
    return {in, out};
}
```

Obviously, creating an overload for counted iterators is unsatisfying.

Option 2: Separate the iterator from the count

This option shows how a library implementer can write just one version of `copy_n` that is automatically optimized for counted iterators. First, we need to provide two utility functions for unpacking and repacking counted iterators:

```
template<WeakIterator I>
I uncounted(I i)
{
    return i;
}

template<WeakIterator I>
I uncounted(counted_iterator<I> i)
{
    return i.it_;
}

template<WeakIterator I>
I recounted(I const &, I i, DifferenceType<I>)
{
    return i;
}

template<WeakIterator I>
counted_iterator<I> recounted(counted_iterator<I> const &j, I i, DifferenceType<I> n)
{
    return {i, j.n_ - n};
}
```

With the help of `uncounted` and `recounted`, we can write an optimized `copy_n` just once:

```
template<WeakInputIterator I, WeakOutputIterator<ValueType<I>> O>
pair<I, O> copy_n(I in_, DifferenceType<I> n_, O out)
{
    auto in = uncounted(in_);
    for(auto n = n_; n ≠ 0; ++in, --n, ++out)
        *out = *in;
```

```
    return {recounted(in_, in, n_), out};
}
```

This version works optimally for both counted and non-counted iterators. It is not a thing of beauty, however. It's slightly annoying to have to do the `uncounted`/`recounted` dance, but it's mostly needed only in the counted algorithms.

As a final note, the overload of `advance` for counted iterators can be eliminated with the help of `uncounted` and `recounted`. After all, `advance` is a counted algorithm.

12.4 Benchmark: Insertion Sort

To test how expensive counted ranges and counted iterators are, we wrote a benchmark. The benchmark program pits counted ranges against a dedicated `_n` algorithm for Insertion Sort.

The program is listed below:

```
#include <chrono>
#include <iostream>
#include <range/v3/range.hpp>

class timer
{
private:
    std::chrono::high_resolution_clock::time_point start_;
public:
    timer()
    {
        reset();
    }
    void reset()
    {
        start_ = std::chrono::high_resolution_clock::now();
    }
    std::chrono::milliseconds elapsed() const
    {
        return std::chrono::duration_cast<std::chrono::milliseconds>(
            std::chrono::high_resolution_clock::now() - start_);
    }
    friend std::ostream &operator<<(std::ostream &sout, timer const &t)
    {
        return sout << t.elapsed().count() << "ms";
    }
};

template<typename It>
struct forward_iterator
{
    It it_;

    template<typename U>
```

```
    friend struct forward_iterator;
public:
    typedef          std::forward_iterator_tag                 iterator_category;
    typedef typename std::iterator_traits<It>::value_type      value_type;
    typedef typename std::iterator_traits<It>::difference_type difference_type;
    typedef It                                                 pointer;
    typedef typename std::iterator_traits<It>::reference       reference;

    forward_iterator() : it_() {}
    explicit forward_iterator(It it) : it_(it) {}

    reference operator*() const {return *it_;}
    pointer operator→() const {return it_;}

    forward_iterator& operator++() {++it_; return *this;}
    forward_iterator operator++(int)
        {forward_iterator tmp(*this); ++(*this); return tmp;}

    friend bool operator==(const forward_iterator& x, const forward_iterator& y)
        {return x.it_ == y.it_;}
    friend bool operator≠(const forward_iterator& x, const forward_iterator& y)
        {return !(x == y);}
};

template<typename I, typename V2>
I upper_bound_n(I begin, typename std::iterator_traits<I>::difference_type d, V2 const &val)
{
    while(0 ≠ d)
    {
        auto half = d / 2;
        auto middle = std::next(begin, half);
        if(val < *middle)
            d = half;
        else
        {
            begin = ++middle;
            d -= half + 1;
        }
    }
    return begin;
}

template<typename I>
void insertion_sort_n(I begin, typename std::iterator_traits<I>::difference_type n)
{
    auto m = 0;
    for(auto it = begin; m ≠ n; ++it, ++m)
    {
        auto insertion = upper_bound_n(begin, m, *it);
        ranges::rotate(insertion, it, std::next(it));
    }
}

template<typename I, typename S>
void insertion_sort(I begin, S end)
{
```

```
    for(auto it = begin; it ≠ end; ++it)
    {
        auto insertion = ranges::upper_bound(begin, it, *it);
        ranges::rotate(insertion, it, std::next(it));
    }
}

template<typename Rng>
void insertion_sort(Rng && rng)
{
    ::insertion_sort(std::begin(rng), std::end(rng));
}

std::unique_ptr<int[]> data(int i)
{
    std::unique_ptr<int[]> a(new int[i]);
    auto rng = ranges::view::counted(a.get(), i);
    ranges::iota(rng, 0);
    return a;
}

void shuffle(int *a, int i)
{
    auto rng = ranges::view::counted(a, i);
    ranges::random_shuffle(rng);
}

constexpr int cloops = 3;

template<typename I>
void benchmark_n(int i)
{
    auto a = data(i);
    long ms = 0;
    for(int j = 0; j < cloops; ++j)
    {
        ::shuffle(a.get(), i);
        timer t;
        insertion_sort_n(I{a.get()}, i);
        ms += t.elapsed().count();
    }
    std::cout << (int)((double)ms/cloops) << std::endl;
}

template<typename I>
void benchmark_counted(int i)
{
    auto a = data(i);
    long ms = 0;
    for(int j = 0; j < cloops; ++j)
    {
        ::shuffle(a.get(), i);
        timer t;
        insertion_sort(ranges::view::counted(I{a.get()}, i));
        ms += t.elapsed().count();
    }
```

```
    std::cout << (int)((double)ms/cloops) << std::endl;
}

int main(int argc, char *argv[])
{
    if(argc < 2)
        return -1;

    int i = std::atoi(argv[1]);
    std::cout << "insertion_sort_n (random-access) : ";
    benchmark_n<int*>(i);
    std::cout << "insertion_sort   (random-access) : ";
    benchmark_counted<int*>(i);
    std::cout << "\n";
    std::cout << "insertion_sort_n (forward)       : ";
    benchmark_n<forward_iterator<int*>>(i);
    std::cout << "insertion_sort   (forward)       : ";
    benchmark_counted<forward_iterator<int*>>(i);
}
```

The program implements both `insertion_sort_n`, a dedicated counted algorithm, and `insertion_sort`, a general algorithm that accepts any Iterable, to which we pass a counted range. The latter is implemented in terms of the general-purpose `upper_bound` as provided by the Range v3 library, whereas the former requires a dedicated `upper_bound_n` algorithm, which is also provided.

The test is run both with raw pointers (hence, random-access) and with an iterator wrapper that only models ForwardIterator. Each test is run three times, and the resulting times are averaged. The test was compiled with `g++` version 4.9.0 with `-O3 -std=gnu++11 -DNDEBUG` and run on a Linux machine. The results are reported below, for N == 30,000:

	insertion_sort_n	**insertion_sort**
random-access	2.692 s	2.703 s
forward	23.853 s	23.817 s

The performance difference, if there is any, is lost in the noise. At least in this case, with this compiler, on this hardware, there is no performance justification for a dedicated `_n` algorithm.

Summary

In short, counted iterators are not a *perfect* abstraction. There is some precedent here. The iterators for `deque`, and for any segmented data structure, are known to be inefficient (see Segmented Iterators and Hierarchical Algorithms, Austern 1998[2]). The fix for that problem, new iterator abstractions and separate hierarchical algorithm implementations, is invasive and is not attempted in any STL implementation we are aware of. In comparison, the extra complications that come with counted iterators seem quite small. For segmented iterators, the upside was the simplicity and uniformity of the Iterator abstraction. In the case of counted ranges and

iterators, the upside is the simplicity and uniformity of the Iterable concept. Algorithms need only one form, not separate bounded, counted, and sentinel forms. Our benchmark gives us reasonable assurance that we aren't sacrificing performance for the sake of a unifying abstraction.

13 Appendix 5: Drive-By Improvements to the Standard Algorithms

As we are making changes to the standard algorithms, not all of which are strictly source compatible, here are some other drive-by changes that we might consider making. The changes suggested below have nothing to do with ranges *per se*, but they increase the power and uniformity of the STL and they have proven useful in the Adobe Source Library, so we might consider taking all these changes in one go.

13.1 Higher-Order Algorithms Should Take Invokables Instead of Functions

Some algorithms like `for_each` are higher-order; they take functions as parameters. In N3351[21], they are constrained with the `Function` concept which, among other things, requires that its parameters can be used to form a valid callable expression `f(a)`.

However, consider a class `S` with a member function `Do`, like:

```
class S {
public:
    void Do() const;
};
```

If we have a `vector` of `S` objects and we want to `Do` all of them, this is what we need to do:

```
for_each( v, [](auto & s) { s.Do(); });
```

or, more concisely with a `bind` expression:

```
for_each( v, bind(&S::Do, _1) );
```

Note that `bind` is specified in terms of a hypothetical INVOKE utility in [func.require]. Wouldn't it be more convenient if all the algorithms were required to merely take INVOKE-able things – that is, things that can be passed to `bind` – as arguments, instead of Functions? Then we can express the above call to `for_each` more concisely as:

```
for_each( v, &S::Do );
```

We can define an `invokable` utility function as:

```
template<typename R, typename T>
auto invokable(R T::* p) const → decltype(std::mem_fn(p))
{
    return std::mem_fn(p);
}

template<typename T, typename U = decay_t<T>>
auto invokable(T && t) const → enable_if_t<!is_member_pointer<U>::value, T>
{
    return std::forward<T>(t);
}

template<typename F>
using invokable_t = decltype(invokable(std::declval<F>()));
```

We can define an Invokable concept as:

```
concept Invokable<Semiregular F, typename... As> =
    Function<invokable_t<F>, As...> &&
    requires (F f, As... as) {
        InvokableResultOf<F, As...>;
        InvokableResultOf<F, As...> == invokable(f)(as...);
    };
```

The Invokable concept can be used to constrain algorithms instead of the Function concept. The algorithms would need to apply `invokable` to each Invokable argument before invoking it.

This is pure extension and would break no code.

13.2 Algorithms Should Take Invokable Projections

The Adobe Source Libraries (ASL)[1] pioneered the use of "projections" to make the algorithms more powerful and expressive by increasing interface symmetry. Sean Parent gives a motivating example in his "C++ Seasoning" talk[15], on slide 38. With today's STL, when using `sort` and `lower_bound` together with user-defined predicates, the predicate must sometimes differ. Consider:

```
std::sort(a, [](const employee& x, const employee& y)
             { return x.last < y.last; });
auto p = std::lower_bound(a, "Parent", [](const employee& x, const string& y)
                                       { return x.last < y; });
```

Notice the different predicates used in the invocations of `sort` and `lower_bound`. Since the predicates are different, there is a chance they might get out of sync leading to subtle bugs.

By introducing the use of projections, this code is simplified to:

```
std::sort(a, std::less<>(), &employee::last);
auto p = std::lower_bound(a, "Parent", std::less<>(), &employee::last);
```

Every element in the input sequence is first passed through the projection `&employee::last`. As a result, the simple comparison predicate `std::less<>` can be used in both places.

No effort was made in ASL to use projections consistently. This proposal bakes them in everywhere it makes sense. Here are the guidelines to be applied to the standard algorithms:

- Wherever appropriate, algorithms should optionally take INVOKE-able *projections* that are applied to each element in the input sequence(s). This, in effect, allows users to trivially transform each input sequence for the sake of that single algorithm invocation.
- Algorithms that take two input sequences should (optionally) take two projections.
- For algorithms that optionally accept functions/predicates (e.g. `transform`, `sort`), projection arguments follow functions/predicates. There are no algorithm overloads that allow the user to specify the projection without also specifying a predicate, even if the default would suffice. This is to reduce the number of overloads and also to avoid any potential for ambiguity.

An open design question is whether all algorithms should take projections, or only the higher-order algorithms. In our current design, all algorithms take projections. This makes it trivial to, say, copy all the keys of a map by using the standard `copy` algorithm with `&pair<const key,value>::first` as the projection.

13.2.1 Projections versus Range Transform View

In a sense, the use of a projection parameter to an algorithm is similar to applying a transform view directly to a range. For example, calling `std::find` with a projection is similar to applying a transform to a range and calling without the projection:

```
auto it = std::find( a, 42, &employee::age );

auto a2 = a | view::transform( &employee::age );
auto it2 = std::find( a2, 42 );
```

Aside from the extra verbosity of the view-based solution, there are two meaningful differences: (1) The type of the resulting iterator is different; `*it` refers to an `employee` whereas `*it2` refers to an `int`. And (2) if the transform function returns an rvalue, then the transformed view cannot model a forward sequence due to the current requirements on the ForwardIterator concept. The result of applying a transform view is an Input range unless the transform function returns an lvalue. The projection-based interface suffers no such degradation of the iterator category. (Note: if the concepts in N3351[21] are adopted, this argument is no longer valid.) For those reasons,

range transform adapters are not a replacement for projection arguments to algorithms.

See Algorithm Implementation with Projections for a discussion of how projections affect the implementation.

The addition of projection arguments to the algorithms is pure extension.

14 Appendix 6: Implementation Notes

14.1 On Distinguishing Ranges from Non-Range Iterables

The design of the range library depends on the ability to tell apart Ranges from Iterables. Ranges are lightweight objects that refer to elements they do not own. As a result, they can guarantee O(1) copyability and assignability. Iterables, on the other hand, may or may not own their elements, and so cannot guarantee anything about the algorithmic complexity of their copy and assign operations. Indeed, an Iterable may not be copyable at all: it may be a native array or a `vector` of move-only types.

But how to tell Ranges apart from Iterables? After all, whether an Iterable owns its elements or not is largely a semantic difference with no purely syntactic way to differentiate. Well, that's almost true…

It turns out that there is a reasonably good heuristic we can use to tell Iterables and Ranges apart. Imagine that we have some Iterable type `T` that is either a container like `list`, `vector`, or a native array; or else it's a Range like `pair<int*,int*>`. Then we can imagine taking iterators to `T` and `const T`, dereferencing them, and comparing the resulting reference types. The following table gives the results we might expect to find.

Expression	**Container**	**Range**
`*begin(declval<T&>())`	`value_type &`	`[const] value_type &`
`*begin(declval<const T&>())`	`const value_type &`	`[const] value_type &`

Notice how containers and ranges differ in the way a top-level cv-qualifier affects the reference type. Since a range is a proxy to elements stored elsewhere, a top-level `const` qualification on the *range* object typically has no effect at all on its iterator's reference type. But that's not true for a container that owns its elements.

We can use this to build an `is_range` traits that gives a pretty good guess whether an Iterable type is a range or not. This trait can be used to define the Range concept. Obviously since it's a trait, users are free to specialize it if the trait guesses wrong.

Some people want their range types to behave like containers with respect to the handling of top-level `const`; that is, they would like their ranges to be designed such that if the range object is `const`, the range's elements cannot be mutated through it. There is nothing about the Range concepts that precludes that design, but it does require the developer of such a range to specialize the `is_range` trait. If anything, the default behavior of the trait can be seen as gentle encouragement to handle top-level `const` in a way that is consistent with ranges' nature as a lightweight proxy.

As an added convenience, we can provide a class, `range_base`, from which users can create a derived type as another way of opting in to "range-ness". The `is_range` trait can test for derivation from `range_base` as an extra test. This would save users the trouble of opening the `std` namespace to specialize the `is_range` trait on the rare occasions that that is necessary.

The `is_range` trait will also need to be specialized for immutable containers, for which both the mutable and const iterators have the same reference type. A good example is `std::set`.

If the `is_range` trait erroneously reports `false` for a type that is actually a range, then the library errs on the side of caution and will prevent the user from using rvalues of that type in range adaptor pipelines. If, on the other hand, the `is_range` trait gets the answer wrong for a type that is actually a container, the container ends up being copied or moved into range adaptors. This is a performance bug, and it may give surprising results at runtime if the original container doesn't get mutated when the user thinks it should. It's not a memory error, though.

14.2 Algorithm Implementation with Projections

Rather than requiring additional overloads, the addition of projection arguments has very little cost to library implementers. The use of function template default parameters obviates the need for overloads. For instance, `find` can be defined as:

```
template<InputIterator I, Regular S, typename V, Invokable<ValueType<I>> Proj = identity>
    requires EqualityComparable<I, S> &&
             EqualityComparable<V, InvokableResultOf<Proj, ValueType<I>>>
I find(I first, S last, V const & val, Proj proj = Proj{})
{
    /* ... */
}
```

14.3 Algorithms That Need An End Iterator

Some algorithms need to know the real physical end of the input sequence so that the sequence can be traversed backwards, like `reverse`. In those cases, it's helpful to have an algorithm `advance_to` that takes an iterator and a sentinel and returns a real end iterator. `advance_to` looks like this:

```
template<Iterator I, Regular S>
    requires IteratorRange<I, S>
I advance_to( I i, S s )
{
    while(i ≠ s)
        ++i;
    return i;
}

template<Iterator I, Regular S>
    requires SizedIteratorRange<I, S>
I advance_to( I i, S s )
{
    advance( i, s - i );
    return i;
}

template<Iterator I>
I advance_to( I, I s )
{
    return s;
}
```

When the sentinel is actually an iterator, we already know where the end is so we can just return it. Notice how we handle SizedIteratorRanges specially and dispatch to `advance` with a count. Appendix 4 shows how `advance` is optimized for counted iterators. By dispatching to `advance` when we can, we make `advance_to` faster for counted iterators, too.

With `advance_to` we can implement `reverse` generically as:

```
template<BidirectionalIterator I, Regular S>
    requires EqualityComparable<I, S> && Permutable<I>
I reverse( I first, S last_ )
{
    I last = advance_to( first, last_ ), end = last;
    while( first ≠ last )
    {
        if( first == --last )
            break;
        iter_swap( first, last );
        ++first;
    }
    return end;
}
```

Since this algorithm necessarily computes the end of the sequence if it isn’t known already, we return it.