Unifying Operator and Function-Object Variants of Standard Library Algorithms

Author: Douglas Gregor, Indiana University
Document number: N2743=08-0253
Date: 2008-08-25
Project: Programming Language C++, Library Working Group
Reply-to: Douglas Gregor <dgregor@osl.iu.edu>

Introduction

This proposal unifies the operator-based and function-object variants of Standard Library algorithms, in many cases collapsing two algorithm declarations into a single declaration. For example,

```
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void sort(Iter first, Iter last);

template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void sort(Iter first, Iter last,
            Compare comp);
```

becomes

```
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare = less<Iter::value_type>>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void sort(Iter first, Iter last,
            Compare comp = Compare());
```

When the final argument to `sort` is omitted, it will receive a default of `std::less` on the iterator's value type. Since `std::less` itself has a function-call operator that uses the `LessThanComparable` concept, we get the same behavior from this version of `sort` that we would get from the original first variant of `sort`.

Some Standard Library implementations already forward the operator-based formulations of their algorithm implementations to the function-object—based formulations, and this idea is far from new. However, it was not possible to unify the declarations in C++03 for two reasons. First, default arguments of function templates were not supported in C++03. Second, it was not clear in C++03 whether the `operator<` used by the first variant of `sort` operated on the iterator's `value_type` or `reference` type (or a combination of both); concepts tie down these details, making it safe to unify the

signatures. This unification significantly reduces the number of declarations in the algorithms chapter, making it easier to read. The result will be an easier-to-digest Standard Library specification with a less-verbose implementation.

Note that the changes to the algorithms are not quite as extensive as one might hope. In cases where we have type-symmetric operator requirements (`LessThanComparable` and `EqualityComparable`) we can use the standard library's function object types (`std::less` and `equal_to`, respectively). However, wherever we have type-asymmetric operator requirements (`HasLess` and `HasEqualTo`), there is no function object type in the standard. However, introducing a new, mostly redundant set of function object types to the standard library just to simplify the presentation of the algorithms does not seem like a reasonable trade-off.

Chapter 25 Algorithms library [algorithms]

Header `<algorithm>` **synopsis**

Update the synopsis to reflect changes to the algorithms detailed below.

25.1 Non-modifying sequence operations [alg.nonmodifying]

25.1.8 Adjacent find [alg.adjacent.find]

```
template<ForwardIterator Iter>
  requires EqualityComparable<Iter::value_type>
  Iter adjacent_find(Iter first, Iter last);

template<ForwardIterator Iter,
         EquivalenceRelation<auto, Iter::value_type> Pred = equal_to<Iter::value_type>>
  requires CopyConstructible<Pred>
  Iter adjacent_find(Iter first, Iter last,
                     Pred pred = Pred());
```

1 *Returns:* The first iterator `i` such that both `i` and `i + 1` are in the range `[`*first*`,`*last*`)` for which the following corresponding conditions holds: ~~`*i == *(i + 1),`~~ *pred*`(*i, *(i + 1)) != false`. Returns *last* if no such iterator is found.

2 *Complexity:* For a nonempty range, exactly `min((i - `*first*`) + 1, (`*last* ` - `*first*`) - 1)` applications of the corresponding predicate, where `i` is `adjacent_find`'s return value.

25.2 Mutating sequence operations [alg.modifying.operations]

25.2.9 Unique [alg.unique]

```
template<ForwardIterator Iter>
  requires OutputIterator<Iter, Iter::reference>
        && EqualityComparable<Iter::value_type>
  Iter unique(Iter first, Iter last);

template<ForwardIterator Iter,
         EquivalenceRelation<auto, Iter::value_type> Pred = equal_to<Iter::value_type>>
  requires OutputIterator<Iter, RvalueOf<Iter::reference>::type>
        && CopyConstructible<Pred>
  Iter unique(Iter first, Iter last,
              Pred pred = Pred());
```

1 *Effects:* For a nonempty range, eliminates all but the first element from every consecutive group of equivalent elements referred to by the iterator `i` in the range `[`*`first`*` + 1,`*`last`*`)` for which the following conditions holds: ~~`*(i - 1) == *i or`~~ *`pred`*`(*(i - 1), *i) != false`.

2 ~~*Requires:* The comparison function shall be an equivalence relation.~~

3 *Returns:* The end of the resulting range.

4 *Complexity:* For nonempty ranges, exactly `(`*`last`*` - `*`first`*`) - 1` applications of the corresponding predicate.

```
template<InputIterator InIter, typename OutIter>
  requires OutputIterator<OutIter, InIter::reference>
        && OutputIterator<OutIter, const InIter::value_type&>
        && EqualityComparable<InIter::value_type>
        && CopyAssignable<InIter::value_type>
        && CopyConstructible<InIter::value_type>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result);
```

```
template<InputIterator InIter, typename OutIter,
         EquivalenceRelation<auto, InIter::value_type> Pred = equal_to<Iter::value_type>>
  requires OutputIterator<OutIter, InIter::reference>
        && OutputIterator<OutIter, const InIter::value_type&>
        && CopyAssignable<InIter::value_type>
        && CopyConstructible<InIter::value_type>
        && CopyConstructible<Pred>
  OutIter unique_copy(InIter first, InIter last,
                      OutIter result, Pred pred = Pred());
```

Note that the unification of the `unique_copy` signatures depends on the simplification to `unique_copy` in N2742. If that proposal is not accepted, this `unique_copy` cannot be collapsed.

5 *Requires:* The ranges `[`*`first`*`,`*`last`*`)` and `[`*`result`*`,`*`result`*`+(`*`last`*`-`*`first`*`))` shall not overlap.

6 *Effects:* Copies only the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[`*`first`*`,`*`last`*`)` for which the following corresponding conditions holds: ~~`*i == *(i - 1) or`~~ *`pred`*`(*i, *(i - 1)) != false`.

7 *Returns:* The end of the resulting range.

8 *Complexity:* For nonempty ranges, exactly *`last`*` - `*`first`*` - 1` applications of the corresponding predicate.

25.3 Sorting and related operations [alg.sorting]

1 All the operations in 25.3 have two versions: one that takes a function object of type `Compare` and one that uses an `operator<`. [*Note:* in some cases, the version that uses `operator<` does so through default function arguments and default template arguments. — *end note*]

7 In the descriptions of the functions that deal with ordering relationships we frequently use a notion of equivalence to describe concepts such as stability. The equivalence to which we refer is not necessarily an `operator==`, but an

equivalence relation induced by the strict weak ordering. That is, two elements `a` and `b` are considered equivalent if and only if ~~`!(a < b) && !(b < a)`~~`!comp(a, b) && !comp(b, a)`.

25.3.1 Sorting [alg.sort]

25.3.1.1 `sort` [sort]

```
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void sort(Iter first, Iter last);

template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare = less<Iter::value_type>>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void sort(Iter first, Iter last,
            Compare comp = Compare());
```

1 *Effects:* Sorts the elements in the range `[first,last)`.

2 *Complexity:* Approximately $N \log(N)$ (where N `== last - first`) comparisons on the average.[1)]

25.3.1.2 `stable_sort` [stable.sort]

```
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void stable_sort(Iter first, Iter last);

template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare = less<Iter::value_type>>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void stable_sort(Iter first, Iter last,
                   Compare comp = Compare());
```

1 *Effects:* Sorts the elements in the range `[first,last)`.

2 *Complexity:* It does at most $N \log^2(N)$ (where N `== last - first`) comparisons; if enough extra memory is available, it is $N \log(N)$.

3 *Remarks:* Stable.

25.3.1.3 `partial_sort` [partial.sort]

```
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
```

[1)] If the worst case behavior is important `stable_sort()` (25.3.1.2) or `partial_sort()` (25.3.1.3) should be used.

```
      && LessThanComparable<Iter::value_type>
  void partial_sort(Iter first,
                    Iter middle,
                    Iter last);
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare = less<Iter::value_type>>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void partial_sort(Iter first,
                    Iter middle,
                    Iter last,
                    Compare comp = Compare());
```

1 *Effects:* Places the first *middle* - *first* sorted elements from the range [*first*,*last*) into the range [*first*,*middle*). The rest of the elements in the range [*middle*,*last*) are placed in an unspecified order.

2 *Complexity:* It takes approximately (*last* - *first*) * log(*middle* - *first*) comparisons.

25.3.1.5 is_sorted [is.sorted]

```
template<ForwardIterator Iter>
  requires LessThanComparable<Iter::value_type>
  bool is_sorted(Iter first, Iter last);
```

1 *Returns:* is_sorted_until(first, last) == last

```
template<ForwardIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare = less<Iter::value_type>>
  requires CopyConstructible<Compare>
  bool is_sorted(Iter first, Iter last,
                 Compare comp = Compare());
```

2 *Returns:* is_sorted_until(first, last, comp) == last

```
template<ForwardIterator Iter>
  requires LessThanComparable<Iter::value_type>
  Iter is_sorted_until(Iter first, Iter last);
template<ForwardIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare = less<Iter::value_type>>
  requires CopyConstructible<Compare>
  Iter is_sorted_until(Iter first, Iter last,
                       Compare comp = Compare());
```

3 *Returns:* If distance(first, last) < 2, returns last. Otherwise, returns the last iterator i in [first,last] for which the range [first,i) is sorted.

4 *Complexity:* Linear.

25.3.2 Nth element [alg.nth.element]

```
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void nth_element(Iter first, Iter nth,
                   Iter last);

template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare = less<Iter::value_type>>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void nth_element(Iter first, Iter nth,
                   Iter last,  Compare comp = Compare());
```

1 After `nth_element` the element in the position pointed to by *nth* is the element that would be in that position if the whole range were sorted. Also for any iterator `i` in the range `[`*first*`,`*nth*`)` and any iterator `j` in the range `[`*nth*`,`*last*`)` it holds that@~~: !(*i > *j) or~~ *comp*`(*j, *i) == false`.

2 *Complexity:* Linear on average.

25.3.4 Merge [alg.merge]

```
template<BidirectionalIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void inplace_merge(Iter first,
                     Iter middle,
                     Iter last);

template<BidirectionalIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare = less<Iter::value_type>>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void inplace_merge(Iter first,
                     Iter middle,
                     Iter last, Compare comp = Compare());
```

6 *Effects:* Merges two sorted consecutive ranges `[`*first*`,`*middle*`)` and `[`*middle*`,`*last*`)`, putting the result of the merge into the range `[`*first*`,`*last*`)`. The resulting range will be in non-decreasing order; that is, for every iterator `i` in `[`*first*`,`*last*`)` other than *first*, the condition ~~*i < *(i - 1) or, respectively,~~ *comp*`(*i, *(i - 1))` will be false.

7 *Complexity:* When enough additional memory is available, `(`*last* `-` *first*`) - 1` comparisons. If no additional memory is available, an algorithm with complexity $N \log(N)$ (where `N` is equal to *last* `-` *first*) may be used.

8 *Remarks:* Stable.

25.3.6 Heap operations [alg.heap.operations]

1 A *heap* is a particular organization of elements in a range between two random access iterators `[a,b)`. Its two key

properties are:

(1) There is no element greater than `*a` in the range and

(2) `*a` may be removed by `pop_heap()`, or a new element added by `push_heap()`, in $\mathcal{O}(\log(N))$ time.

2 These properties make heaps useful as priority queues.

3 `make_heap()` converts a range into a heap and `sort_heap()` turns a heap into a sorted sequence.

25.3.6.1 `push_heap` [push.heap]

```
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void push_heap(Iter first, Iter last);

template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare = less<Iter::value_type>>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void push_heap(Iter first, Iter last,
                 Compare comp = Compare());
```

1 *Effects:* Places the value in the location `last - 1` into the resulting heap `[first, last)`.

2 *Requires:* The range `[first, last - 1)` shall be a valid heap.

3 *Complexity:* At most `log(last - first)` comparisons.

25.3.6.2 `pop_heap` [pop.heap]

```
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter> && LessThanComparable<Iter::value_type>
  void pop_heap(Iter first, Iter last);

template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare = less<Iter::value_type>>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void pop_heap(Iter first, Iter last,
                Compare comp = Compare());
```

1 *Effects:* Swaps the value in the location `first` with the value in the location `last - 1` and makes `[first, last - 1)` into a heap.

2 *Requires:* The range `[first, last)` shall be a valid heap.

3 *Complexity:* At most `2 * log(last - first)` comparisons.

25.3.6.3 make_heap [make.heap]

```
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void make_heap(Iter first, Iter last);

template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare = less<Iter::value_type>>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void make_heap(Iter first, Iter last,
                 Compare comp = Compare());
```

1 *Effects:* Constructs a heap out of the range [*first*, *last*).

2 *Complexity:* At most `3 * (last - first)` comparisons.

25.3.6.4 sort_heap [sort.heap]

```
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter> && LessThanComparable<Iter::value_type>
  void sort_heap(Iter first, Iter last);

template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare = less<Iter::value_type>>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void sort_heap(Iter first, Iter last,
                 Compare comp = Compare());
```

1 *Effects:* Sorts elements in the heap [*first*, *last*).

2 *Complexity:* At most $N \log(N)$ comparisons (where `N == last - first`).

25.3.6.5 is_heap [is.heap]

```
template<RandomAccessIterator Iter>
  requires LessThanComparable<Iter::value_type>
  bool is_heap(Iter first, Iter last);
```

1 *Returns:* `is_heap_until(first, last) == last`

```
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare = less<Iter::value_type>>
  requires CopyConstructible<Compare>
  bool is_heap(Iter first, Iter last, Compare comp = Compare());
```

2 *Returns:* `is_heap_until(first, last, comp) == last`

```
template<RandomAccessIterator Iter>
  Iter is_heap_until(Iter first, Iter last);
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare = less<Iter::value_type>>
  requires CopyConstructible<Compare>
  Iter is_heap_until(Iter first, Iter last,
                     Compare comp = Compare());
```

3 *Returns:* If `distance(first, last) < 2`, returns `last`. Otherwise, returns the last iterator `i` in `[first,last]` for which the range `[first,i)` is a heap.

4 *Complexity:* Linear.

25.3.7 Minimum and maximum [alg.min.max]

```
template<ForwardIterator Iter>
  requires LessThanComparable<Iter::value_type>
  Iter min_element(Iter first, Iter last);

template<ForwardIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare = less<Iter::value_type>>
  requires CopyConstructible<Compare>
  Iter min_element(Iter first, Iter last,
                   Compare comp = Compare());
```

31 *Returns:* The first iterator `i` in the range `[`*first*`,`*last*`)` such that for any iterator `j` in the range `[`*first*`,`*last*`)` the following corresponding conditions holds: `!(*j < *i)` or *comp*`(*j, *i) == false`. Returns *last* if *first* `==` *last*.

32 *Complexity:* Exactly `max((`*last* `-` *first*`) - 1, 0)` applications of the corresponding comparisons.

```
template<ForwardIterator Iter>
  requires LessThanComparable<Iter::value_type>
  Iter max_element(Iter first, Iter last);

template<ForwardIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare = less<Iter::value_type>>
  requires CopyConstructible<Compare>
  Iter max_element(Iter first, Iter last,
                   Compare comp = Compare());
```

33 *Returns:* The first iterator `i` in the range `[`*first*`,`*last*`)` such that for any iterator `j` in the range `[`*first*`,`*last*`)` the following corresponding conditions holds: `!(*i < *j)` or *comp*`(*i, *j) == false`. Returns *last* if *first* `==` *last*.

34 *Complexity:* Exactly `max((`*last* `-` *first*`) - 1, 0)` applications of the corresponding comparisons.

```
template<ForwardIterator Iter>
  requires LessThanComparable<Iter::value_type>
  pair<Iter, Iter>
    minmax_element(Iter first, Iter last);
```

```
template<ForwardIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare = less<Iter::value_type>>
  requires CopyConstructible<Compare>
  pair<Iter, Iter>
    minmax_element(Iter first, Iter last, Compare comp = Compare());
```

35 *Returns:* `make_pair(m, M)`, where `m` is ~~`min_element(first, last) or`~~ `min_element(first, last, comp)` and `M` is ~~`max_element(first, last) or`~~ `max_element(first, last, comp)`.

36 *Complexity:* At most `max(2 * (last - first) - 2, 0)` applications of the corresponding comparisons.

25.3.9 Permutation generators [alg.permutation.generators]

```
template<BidirectionalIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  bool next_permutation(Iter first, Iter last);

template<BidirectionalIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare = less<Iter::value_type>>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  bool next_permutation(Iter first, Iter last, Compare comp = Compare());
```

1 *Effects:* Takes a sequence defined by the range `[first, last)` and transforms it into the next permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to ~~`operator< or`~~ *comp*. If such a permutation exists, it returns `true`. Otherwise, it transforms the sequence into the smallest permutation, that is, the ascendingly sorted one, and returns `false`.

2 *Complexity:* At most `(last - first)/2` swaps.

```
template<BidirectionalIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  bool prev_permutation(Iter first, Iter last);

template<BidirectionalIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare = less<Iter::value_type>>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  bool prev_permutation(Iter first, Iter last, Compare comp = Compare());
```

3 *Effects:* Takes a sequence defined by the range `[first, last)` and transforms it into the previous permutation. The previous permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to ~~`operator< or`~~ *comp*.

4 *Returns:* `true` if such a permutation exists. Otherwise, it transforms the sequence into the largest permutation, that is, the descendingly sorted one, and returns `false`.

5 *Complexity:* At most `(last - first)/2` swaps.