Doc No: N2569=08-0079
Reply to: Matt Austern <austern@google.com>
Date: 2008-02-29

More STL algorithms

This paper proposes a number of nonstandard STL-style algorithms for inclusion in the standard. Nothing in this paper is novel or complicated. All of these algorithms have been around for years, and have been described in books and provided by library vendors as nonstandard extensions. All of them are easily implementable, and have been implemented. Most of them are taken from the SGI STL (http://www.sgi.com/tech/stl).

This proposal is a pure extension.

Design Decisions

In most cases there aren't any very important decisions to be made once we've decided to provide the algorithm in the first place.

Counting versions of copy

The standard `copy` algorithm takes three arguments: a first/last input iterator pair to determine the input range, and a single output iterator to determine the start of the output range. Sometimes, however, it's more convenient to describe the operation as a starting point and a count than as a starting point and an ending point. This is usually just a matter of convenience, since we can usually convert between those forms by writing `n = distance(first, last)` or `last = first, advance(last, n)`, but it can occasionally be more important if the iterators are of a type where converting between a range and a count would be expensive or impractical. (Input iterators that aren't forward iterators, for example.) The SGI STL provides `copy_n` and the corresponding version for initializing raw memory, `uninitialized_copy_n`.

All, any, and none

These three algorithms provide the ordinary mathematical operations $\forall$, $\exists$, and $\nexists$: given a range and a predicate, determine whether that predicate is true for all elements; whether there exists an element for which the predicate is true; or whether there exist no elements for which the predicate is true. Strictly speaking there is no need to provide all three of these algorithms (`!none` and `any` are equivalent), but all three of these operations feel equally fundamental.

These three algorithms have accumulated various names over the years. I have taken the names `any`/`none`/`all` from the latest draft of Alex Stepanov and Paul McJones's *Elements of Programming*.

copy_if

This is a frequently requested addition, mentioned in (for example) the latest edition of *The C++ Programming Language*. It is formally redundant, since it's just the inverse of `remove_copy_if`: copying all elements that satisfy a predicate `p` is just the same as not copying all elements that satisfy `!p`. It's worth adding anyway. First, C++ isn't really a functional language and transforming a predicate into its negation is sometimes awkward. Second, the workaround of using double negatives is not unconfusing.

find_if_not
Just as `copy_if` is the inverse of `remove_copy_if`, `find_if_not` is the inverse of `find_if`: it returns an iterator pointing to the first element that fails to satisfy a predicate `p`. It's worth adding for the same reason.

Random sampling
The standard contains `random_shuffle`, which randomly permutes a range with uniform distribution. A closely related operation is randomly selecting elements from a range. (Indeed, Knuth discusses those operations in a single subsection, "Random Sampling and Shuffling".) There are two important versions of random sampling, one of which randomly chooses *n* elements from a range of *N* elements and the other of which randomly chooses *n* elements from an input range whose size is initially unknown except that it is at least *n*. These two algorithms are not redundant, even theoretically. Knuth calls them "Algorithm S" and "Algorithm R". The SGI STL, which has provided them for many years, calls them `random_sample` and `random_sample_n`.

Three-way lexicographical comparison
The standard lexicographical comparison algorithm throws away information: if we know that `lexicographical_compare(f1, l1, f2, l2)` returns false, it could mean either that the second range is lexicographically less than the other or that the two are lexicographically equal. Given the existing interface, the only way for the user to get that extra information is with a second linear-time range traversal. Changing the interface, however, allows the lexicographical comparison to compute that extra information with just a single extra comparison.

There are several possible way to of providing the extended version of lexicographical comparison. The SGI STL provides it as `lexicographical_compare_3way`, which uses the ordinary `strcmp` convention: the return value is negative if the first range is less than the second, zero if they are equal, positive if the first range is greater.

Partition algorithms
The original HP STL included `partition`, but not `partition_copy`, mainly because it wasn't clear what the `partition_copy` interface should be: how can you have a single output range when you're partitioning elements into two sets? T. K. Lakshman, then at SGI, pointed out what now seems obvious: `partition_copy` should copy to two output ranges.

Partitioning a range imposes a structure on it. The standard now provides `is_sorted` and `is_heap` to test whether ranges have specific structures, and `is_partitioned` is useful for just the same reasons.

One final partition algorithm is `partition_point`, which takes a partitioned range as input and finds the boundary between the elements that do and don't satisfy the partitioning predicate. This is just a variation of binary search. It generalizes the more common version of binary search, since searching for a value in a sorted range simply means finding the partition point between elements greater and less than that value. In fact, as Dave Abrahams pointed out, this is the most natural way to think about the standard binary search algorithms in the presence of heterogeneous comparisons. (And the wording in the C++0x draft already reflects that insight.)

Iota
This algorithm is inspired by the APL ι operator. Like ι, the `iota` algorithm creates a range of sequentially increasing values. It's especially useful for testing, since permutations of `{1, 2, ... N}` often make good test input, and it has been in the SGI STL for years.

Exponentiation
The SGI STL provides one more nonstandard algorithm, `power`. I am not proposing `power` for standardization at this time because it is too closely tied into concepts that don't yet exist in the standard, `Monoid` and `Semigroup`. The interface for `power` would look different depending on which of those concepts we want it to operate on, and, if we chose `Monoid`, we would have to standardize a mechanism for computing a monoid's identity element. The SGI STL addresses those questions, but not in a completely satisfactory way. More complete mechanisms are described in the draft of *Elements of Programming*.

Proposed Wording

Add the following text to an appropriate subsection of 20.6.4 [specialized.algorithms], and add the signatures to the header <memory> synopsis at the beginning of clause 20.6 [memory].

```
template <class InputIterator, class Size, class ForwardIterator>
  ForwardIterator uninitialized_copy_n(InputIterator first, Size n,
ForwardIterator result);
```

Effects:

```
for ( ; n > 0; ++result, ++first, --n) {
  new (static_cast<void*>(&*result))
    typename iterator_traits<ForwardIterator>::value_type(*first);
}
```

Returns: `result`

Add the following text to appropriate subsections of 25.1 [alg.nonmodifying], and add the signatures to the header `<algorithm>` synopsis at the beginning of clause 25 [algorithms].

```
template <class InputIterator, class Predicate>
  bool all(InputIterator first, InputIterator last, Predicate pred);
```

Returns: `true` if `pred(*i)` is `true` for every iterator in the range `[first, last)`, and `false` otherwise.
Complexity: At most `last - first` applications of the predicate.

```
template <class InputIterator, class Predicate>
  bool any(InputIterator first, InputIterator last, Predicate pred);
```

Returns: `true` if there exists any iterator in the range `[first, last)` such that `pred(*i)` is `true`, and `false` otherwise.
Complexity: At most last - first applications of the predicate.

```
template <class InputIterator, class Predicate>
  bool none(InputIterator first, InputIterator last, Predicate pred);
```

Returns: `true` if `pred(*i)` is `false` for every iterator in the range `[first, last)`, and `false` otherwise.
Complexity: At most `last - first` applications of the predicate.

```
template <class InputIterator, class Predicate>
  bool find_if_not(InputIterator first, InputIterator last, Predicate pred);
```

Returns: the first iterator `i` in the range `[first, last)` such that `pred(*i)` is `false`, or `last` if there is no such iterator.
Complexity: At most last - first applications of the predicate.

Add the following text to appropriate subsections of 25.2 [alg.modifying.operators], and add the signatures to the header `<algorithm>` synopsis at the beginning of clause 25 [algorithms].

```
template <class InputIterator, class Size, class OutputIterator>
  OutputIterator copy_n(InputIterator first, Size n, OutputIterator result);
```

Effects: Copies elements in the range `[first, first+n)` into the range `[result, result+n)` starting from `first` and proceeding to `first+n`. For each non-negative integer `i<n`, performs `*(result+i) = *(first+i)`.
Returns: `result+n`.
Complexity: Exactly `n` assignments.

```
template <class InputIterator, class OutputIterator, class Predicate>
  OutputIterator copy_if(InputIterator first, InputIterator last,
                         OutputIterator result, Predicate pred);
```

Requires: The ranges [first, last) and [result, result + (last - first)) shall not overlap.
Effects: Copies all of the elements referred to by the iterator i in the range [first, last) for which pred(*i) is true.
Returns: The end of the resulting range.
Complexity: Exactly last - first applications of the corresponding predicate.
Remarks: Stable.

```
template <class InputIterator, class RandomAccessIterator>
  RandomAccessIterator random_sample(InputIterator first, InputIterator last,
                                     RandomAccessIterator ofirst,
RandomAccessIterator olast);

template <class InputIterator, class RandomAccessIterator, class
```

```
RandomNumberGenerator>
  RandomAccessIterator random_sample(InputIterator first, InputIterator last,
                                     RandomAccessIterator ofirst,
RandomAccessIterator olast,
                                     RandomNumberGenerator& rand);

template <class InputIterator, class RandomAccessIterator, class
UniformRandomNumberGenerator>
  RandomAccessIterator random_sample(InputIterator first, InputIterator last,
                                     RandomAccessIterator ofirst,
RandomAccessIterator olast,
                                     UniformRandomNumberGenerator& g);
```

Requires: `InputIterator`'s value type shall be Assignable, and shall be convertible to `RandomAccessIterator`'s value type. `RandomNumberGenerator` shall have a return type that is convertible to `InputIterator`'s difference type, and the call `rand(n)` shall return a randomly chosen value in the interval `[0, n)`. The function object `g` shall meet the requirements of a uniform random number generator. `[first, last)` and `[ofirst, olast)` shall be valid and non-overlapping ranges.
Effects: copies `min(last-first, olast-ofirst)` elements from `[first, last)` to `[ofirst, olast)`, such that each possible sampling has equal probability of appearance.
Returns: The end of the resulting range.
Complexity: Linear in `last-first`.
Remarks: Relative order within the input range is not preserved. The underlying sources of randomness is as described for `random_shuffle`.

```
template <class ForwardIterator, class OutputIterator, class Distance>
  OutputIterator random_sample_n(ForwardIterator first, ForwardIterator last,
                                 OutputIterator out, Distance n);

template <class ForwardIterator, class OutputIterator, class Distance,
          class RandomNumberGenerator>
  OutputIterator random_sample_n(ForwardIterator first, ForwardIterator last,
                                 OutputIterator out, Distance n,
                                 RandomNumberGenerator& rand);

template <class ForwardIterator, class OutputIterator, class Distance,
          class UniformRandomNumberGenerator>
  OutputIterator random_sample_n(ForwardIterator first, ForwardIterator last,
                                 OutputIterator out, Distance n,
                                 UniformRandomNumberGenerator& rand);
```

Requires: `ForwardIterator`'s value type shall be Assignable, and shall be writable to the `out` OutputIterator. The input and output ranges shall not overlap. `RandomNumberGenerator` shall have a return type that is convertible to `ForwardIterator`'s difference type, and the call `rand(n)` shall return a randomly chosen value in the interval `[0, n)`. The function object `g` shall meet the requirements of a uniform random number generator.

Effects: copies `min(last-first, n)` elements from `[first, last)` to `out`, such that each possible sampling has equal probability of appearance.
Returns: The end of the resulting range.
Complexity: Linear in `last-first`.
Remarks: Stable. The underlying sources of randomness is as described for `random_shuffle`.

```
template <class InputIterator, class OutputIterator1, class OutputIterator2,
class Predicate>
  pair<OutputIterator1, OutputIterator2>
  partition_copy(InputIterator first, InputIterator last,
                 OutputIterator1 out_true, OutputIterator2 out_false,
                 Predicate pred);
```

Requires: `InputIterator`'s value type shall be Assignable, and shall be writable to the `out_true` and `out_false` OutputIterators, and shall be convertible to `Predicate`'s argument type. The input range shall not overlap with either of the output ranges.
Effects: For each iterator `i` in `[first, last)`, copies `*i` to the output range beginning with `out_true` if `pred(*i)` is `true`, or to the output range beginning with `out_false` otherwise.
Returns: A pair `p` such that `p.first` is the end of the output range beginning with `out_true` and `p.second` is the end of the output range beginning with `out_false`.
Complexity: Exactly `last-first` applications of `pred`.

```
template <class InputIterator, class Predicate>
  bool is_partitioned(InputIterator first, InputIterator last, Predicate
pred);
```

Requires: `InputIterator`'s value type shall be convertible to `Predicate`'s argument type.
Returns: `true` if `[first, last)` is partitioned by `pred`, i.e. if all elements that satisfy `pred` appear before those that do not.
Complexity: Linear. At most `last-first` applications of `pred`.

```
template <class ForwardIterator, class Predicate>
  ForwardIterator partition_point(ForwardIterator first, ForwardIterator
last, Predicate pred);
```

Requires: `ForwardIterator`'s value type shall be convertible to `Predicate`'s argument type. `[first, last)` shall be partitioned by `pred`, i.e. all elements that satisfy `pred` shall appear before those that do not.
Returns: An iterator `mid` such that `all(first, mid, pred)` and `none(mid, last, pred)` are both true.
Complexity: *O*`(log(last-first))` applications of `pred`.

Add the following text to an appropriate subsection of 25.3 [alg.sorting], and add the signature to the header `<algorithm>` synopsis at the beginning of clause 25 [algorithms].

```
template <class InputIterator1, class InputIterator2>
  int lexicographical_compare_3way(InputIterator1 first1, InputIterator1
last2,
                                   InputIterator2 first2, InputIterator2
last2);
```

Returns: A negative number if the sequence of elements defined by the range `[first1, last1)` is lexicographically less than the sequence of elements defined by the range `[first2, last2)`, a positive number if `[first2, last2)` is lexicographically less than `[first1, last1)`, and zero if neither sequence is less than the other.
Complexity: At most 2×min(`last1-first1`, `last2-first2`) comparisons.
Remarks: The definition of lexicographical ordering is the same as for `lexicographical_compare`.

Add the following text as a new subsection of 26.6 [numeric.ops], and add the signature to the header `<numeric>` synopsis at the beginning of clause 26.6 [numeric.ops].

```
template <class ForwardIterator, class T>
  void iota(ForwardIterator first, ForwardIterator last, T value);
```

Requires: `T` shall meet the requirements of CopyConstructible and Assignable types, and shall be convertible to `Forwarditerator`'s value type. `T` shall provide an increment operator.
Effects: For each element referred to by the iterator `i` in the range `[first, last)`, assigns `*i = value` and increments `value` as if by `++value`.
Complexity: Exactly `last-first` increments and assignments.

References

Donald Knuth, *The Art of Computer Programming (Third Edition). Volume 2: Seminumerical Algorithms*.
SGI, *Standard Template Library Programmer's Guide*, http://www.sgi.com/tech/stl
Alex Stepanov and Paul McJones, *Elements of Programming* (draft), http://www.stepanovpapers.com/eop/lecture_all.pdf
Bjarne Stroustrup, *The C++ Programming Language (Special Edition)*.