Algorithms for permutations and combinations, with and without repetitions

Document number:	N2639=08-0149
Document title:	Algorithms for permutations and combinations, with and without repetitions
Author:	Hervé Brönnimann
Contact:	hbr@poly.edu
Organization:	Polytechnic University and Bloomberg L.P.
Date:	2008-5-16
Number:	WG21/N2639
Working Group:	Library

Abstract

This proposal adds eight algorithms (`std::next_partial_permutation`, `next_combination`, `next_mapping`, `next_repeat_combination_counts`, their counterparts `std::prev_partial_permutation`, `std::prev_combination`, `std::prev_mapping`, `std::prev_repeat_combination_counts`, with their overloads) to the header `<algorithm>`, for enumerating permutations and combinations, with and without repetitions. They mirror and extend `std::next_permutation` and `std::prev_permutation`. For sizes known at compile-time, these algorithms can generally be simulated by a number of nested loops.

1 Motivation and Scope

1 This proposal addresses missing functionality in the standard. The standard offers full permutations of a range (with `std::next_permutation`), but not partial permutations or combinations (i.e., selection of a subset of a given size without replacement, also called without repetitions, where order matters or not). These are standard concepts in combinatorics, and they can be enumerated easily when the size is known at compile time. For instance, for enumerating every triplet where order does matter (permutation of size 3 of `[first,last]`), the following nested loops will do:

```
template <typename RandomAccessIter, typename Functor>
void for_each_triplet(RandomAccessIter first,
                      RandomAccessIter last,
                      Functor f)
```

```
{
  for (RandomAccessIter i = first; i != last; ++i) {
      for (RandomAccessIter j = first; j != last; ++j) {
          if (i == j) continue;
          for (RandomAccessIter k = first; k != last; ++k) {
              if (i == k || j == k) continue;
              f(i, j, k);
          }
      }
  }
}
```

while, if order does *not* matter (i.e., a triplet should be enumerated once, not once for each of its six permutations), the following code will do:

```
template <typename RandomAccessIter, typename Functor>
void for_each_3_subset(RandomAccessIter first,
                       RandomAccessIter last,
                       Functor f)
{
  for (RandomAccessIter i = first; i != last; ++i) {
      for (RandomAccessIter j = i + 1; j != last; ++j) {
          for (RandomAccessIter k = j + 1; k != last; ++k) {
              f(i, j, k);
          }
      }
  }
}
```

When the size of the subset (i.e., the number of nested loops) is not known at compile time, however, the implementation becomes a lot harder to get right.

2 We propose to add algorithms to complement `std::next_permutation`: we wish to cover (for completeness) permutations and combinations without and with repetitions. Formally [1,3]:

- A *permutation* of size r of a range of size n is a (not necessarily) sorted subsequence of size r of the total range, i.e., a subsequence of elements at r positions among the n positions in the range.
- A *combination* of size r of a range of size n is a sorted subsequence of size r of the total range, i.e., the ordered (possibly multi-)set of the elements at r positions among the n positions in the range.
- A permutation or combination is *without repetition* if the r indices in the respective definition are distinct (and necessarily $r \leq n$), and *with repetition* otherwise.

3 Combinations and permutations are very useful objects, when you need them, in particular for testing (enumerating all possible testing cases) and for cheaply deriving brute-force combinatorial algorithms [4]. They are also interesting for indexing multi-dimensional data structures (multi-array, data cubes, etc.). In particular, algebraic com-

putations like determinants and permanents, and by extension Grassmann algebra, may rely on the enumeration of such subsets in a specific order.

4 We do provide examples in this proposal, although they tend to be rather lengthy. See section 6 for an extensive example.

5 The algorithms we propose (for consistency and ease of use) follow the same interface as `std::next_permutation`. Unlike the hand-coded loops above, but like `std::next_permutation` does, they also correctly handle multiple values in the input range; for instance, if all values compare equal, there is only one combination and one permutation for any size. As an example of usage, with the appropriate `#include`s, we can generalize the two functions above to arbitrary runtime sizes:

```
template <typename RandomAccessIter, typename Functor>
void for_each_tuple(RandomAccessIter first,
                    RandomAccessIter middle,
                    RandomAccessIter last, Functor f)
{
    std::sort(first, last);
    do {
       f(first, middle);
    } while (next_partial_permutation(first, middle, last));
}

template <typename RandomAccessIter, typename Functor>
void for_each_subset(RandomAccessIter first,
                     RandomAccessIter middle,
                     RandomAccessIter last, Functor f)
{
    std::sort(first, last);
    do {
       f(first, middle);
    } while (next_combination(first, middle, last));
}
```

2 Impact On the Standard

This proposal defines connected but independent pure library extensions. The committee may consider any pair of `next/prev` algorithms separately. That is, the proposal can be split into four distinct proposals which can be considered independently:

— `next/prev_partial_permutation`
— `next/prev_combination`
— `next/prev_mapping`
— `next/prev_repeat_combination_counts`

Their addition does not interact with other parts of the standard that I can think of.

3 Design Decisions

1 There are many possible definitions, and we stuck to the most established and least likely to cause conflict or confusion [1, 3]. Many other variants, e.g., where values are required to be unique, can be easily derived from the algorithms we propose here by additional pre- and post-processing. For instance, requiring all permutations/combinations of size r without repetitions from a range of size n in which all elements are unique (even though the elements in the range may not be) is no different from first removing all duplicates (using `std::unique`) and, if the resulting size is $m < r$, enumerating all the permutations/combinations of r elements out of these m unique elements. Some other variants are harder (see next section), and we do not propose them for standardization either.

2 We designed the interface to be most efficient. Requiring the range `[middle, last)` to be sorted is a good compromise because it does not force the algorithm to do it (unnecessarily so if this is already the case), it makes the algorithm more efficient and is maintained by the natural implementation of the algorithm, and it is easy enough for the user to precondition the range to meet the requirement (via standard algorithms).

3 We also provide generic variants as overload of `next_mapping` and `prev_mapping`, accepting an additional functor (the incrementor) used to increment the value of the mapping. We include them into this proposal for completeness (since we also include overloads of the permutations and combination without repetitions for comparison functors).

4 We decided to not require an explicit representation of the combinations with repetitions, in order to avoid (1) potentially expensive multiple copies of the same repeated instance, and (2) algorithmic costs of merely counting the number of repetitions of an instance at every application of the algorithms. The representation we propose instead, simply gives the multiplicities of each element, hence the `..._counts` name scheme.

5 The names `..._without_repetitions` or `..._with_repetitions` would just be too long for standardization. The permutations of size $r < n$ are still called permutations, but we named the function with the idiom *partial permutations* because "partial" is already used in the C++ standard (`partial_sort`, with the same interface: `(first,middle,last)`), and to avoid overloading ambiguities with the existing `next_permutation` algorithm for certain template parameters. We could have kept the same names for the versions with repetitions, except that the most useful interfaces did not manipulate these permutations or combinations with repetitions per se, but rather the mappings or combination counts, hence the naming.

6 The reference implementation included below convinced us of the usefulness of this proposal, given that the implementation can be succinct and very efficient, but non-trivial, and would take valuable resource to re-derive and test for correctness by each developer.

4 Possible extensions

We did not cover in this proposal the following variants. We personally feel that they belong in a more specialized library (e.g., a future Boost.permutation library [5]), but offer them here to show the richness of the topic:

— `next/prev_cyclic_permutation`: all cyclic permutations are considered equivalent. If any element is distinct, say `*first`, simply enumerate all the permutations of `[++first,last)` keeping `*first` fixed. Complications occur, however, if `*first` is repeated, or if cyclic permutations must be enumerated in lexicographical order and the minimum element is repeated, and the corresponding algorithm becomes highly non-trivial.

— `next/prev_partial_cyclic_permutation`: same with permutations of r among n elements. I'm not sure how complicated the algorithm is... but it's certainly no longer a simple extension like the full cyclic permutation. For one thing, fixing every first element of the partial permutation to each element in turn will enumerate partial permutations twice. I think requiring the first element of the partial permutation to be the minimum of these, thus taking every iterator middle in the range `[first, last-r)` and enumerating the partial permutations of `[++middle, last)` of size $r-1$ would enumerate them in lexicographic order, if all the elements are distinct. Not sure what happens with equal elements, perhaps just skipping over the middle such that `*middle == *(middle-1)` might do the trick. In any case, while easy to describe, this is not an easy algorithm.

— `next/prev_reverse_permutation`: a permutation and its reversal are considered equivalent. The difficulty here seems not to enumerate them (you could skip a permutation if, e.g., the first is greater than the last element, or if the permutation is greater than its reversal if you also wish to take care of equal values). The difficulty seems to enumerate them in lexicographic order with at most `(last - first)` swaps and comparisons (the previously proposed algorithm has no upper bound on the number of operations needed to obtain the next reverse permutation). I think a tweaked implementation of `next_permutation` must be written, it doesn't seem possible to just combine the existing algorithms. For that reason, it would deserve to be a standalone algorithm. Minor point: the name is not a good one, must come up with a better one.

— `next/prev_partial_reverse_permutation`: same with permutations of r among n elements.

Just to be clear, once again, we are not proposing to standardize these extensions.

5 Proposed Text

In the **header <algorithm> synopsis**, rename section 25.3.9 (currently: "permutations") to "combinatorial enumeration", keep the `next_permutation` and `prev_permutation`, and add:

```
// 25.3.9, combinatorial enumeration:
// . . .
template <class BidirectionalIterator>
  bool next_partial_permutation(BidirectionalIterator first,
                                BidirectionalIterator middle,
                                BidirectionalIterator last);
template <class BidirectionalIterator, class Compare>
  bool next_partial_permutation(BidirectionalIterator first,
                                BidirectionalIterator middle,
                                BidirectionalIterator last, Compare comp);

template <class BidirectionalIterator>
  bool prev_partial_permutation(BidirectionalIterator first,
                                BidirectionalIterator middle,
                                BidirectionalIterator last);
template <class BidirectionalIterator, class Compare>
  bool prev_partial_permutation(BidirectionalIterator first,
                                BidirectionalIterator middle,
                                BidirectionalIterator last, Compare comp);

template <class BidirectionalIterator>
  bool next_combination(BidirectionalIterator first,
                        BidirectionalIterator middle,
                        BidirectionalIterator last);
template <class BidirectionalIterator, class Compare>
  bool next_combination(BidirectionalIterator first,
                        BidirectionalIterator middle,
                        BidirectionalIterator last, Compare comp);

template <class BidirectionalIterator>
  bool prev_combination(BidirectionalIterator first,
                        BidirectionalIterator middle,
                        BidirectionalIterator last);
template <class BidirectionalIterator, class Compare>
  bool prev_combination(BidirectionalIterator first,
                        BidirectionalIterator middle,
                        BidirectionalIterator last, Compare comp);

template <class BidirectionalIterator, class T>
  bool next_mapping(BidirectionalIterator first,
                    BidirectionalIterator last,
                    T first_value, T last_value);
template <class BidirectionalIterator, class T, class Incrementor>
  bool next_mapping(BidirectionalIterator first,
                    BidirectionalIterator last,
                    T first_value, T last_value, Incrementor increment);

template <class BidirectionalIterator, class T>
  bool prev_mapping(BidirectionalIterator first,
                    BidirectionalIterator last,
```

```
                    T first_value, T last_value);
template <class BidirectionalIterator, class T, class Decrementor>
  bool prev_mapping(BidirectionalIterator first,
                    BidirectionalIterator last,
                    T first_value, T last_value, Decrementor decrement);

template <class BidirectionalIterator>
  bool next_repeat_combination_counts(BidirectionalIterator first,
                                      BidirectionalIterator last);

template <class BidirectionalIterator>
  bool prev_repeat_combination_counts(BidirectionalIterator first,
                                      BidirectionalIterator last);
```

In section 25.3,9 [alg.permutation.generators], add:

```
template <class BidirectionalIterator>
  bool next_partial_permutation(BidirectionalIterator first,
                                BidirectionalIterator middle,
                                BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
  bool next_partial_permutation(BidirectionalIterator first,
                                BidirectionalIterator middle,
                                BidirectionalIterator last, Compare comp);
```

8 **Effects:** `next_partial_permutation` takes a sequence defined by the range `[first,last)` such that `[first,middle)` stores a *partial permutation*, i.e., a permutation of some subsequence of `[first,last)`, and permutes it such that `[first,middle)` stores the next partial permutation of the same size from `[first,last)`, and `[middle,last)` is sorted. The next partial permutation is found by assuming that the set of all partial permutations of a given size from `[first,last)` is lexicographically sorted with respect to `operator<` or `comp`. If the next partial permutation does not exist, it transforms `[first,middle)` into the smallest partial permutation, leaving the entire range sorted.

9 **Returns:** `true` if the next partial permutation exists, `false` otherwise.

10 **Requires:** The type of `*first` shall satisfy the `Swappable` requirements (37). The range `[middle,last)` shall be sorted in ascending order.

11 **Remarks:** Upon returning false, `[first,middle)` is back to the smallest partial permutation, that is, the prefix of the ascendingly sorted range, and the requirements met for another application of `next_partial_permutation`.

12 **Complexity:** At most `(last - first)` comparisons and `(last - first)` swaps.

13 [*Note:* In order to prepare the range `[first,last)` for an enumeration of all partial permutations in lexicographic order, `std::sort(first,last)` or `std::sort(first,last,comp)`. — *end Note*]

```
template <class BidirectionalIterator>
  bool prev_partial_permutation(BidirectionalIterator first,
```

```
                                BidirectionalIterator middle,
                                BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
  bool prev_partial_permutation(BidirectionalIterator first,
                                BidirectionalIterator middle,
                                BidirectionalIterator last, Compare comp);
```

14 **Effects:** `prev_partial_permutation` takes a sequence defined by the range `[first,last)` such that `[first,middle)` stores a *partial permutation*, i.e., a permutation of some subsequence of `[first,last)`, and permutes it such that `[first,middle)` stores the previous partial permutation of the same size from `[first,last)`, and `[middle,last)` is sorted. The previous partial permutation is found by assuming that the set of all partial permutations of a given size from `[first,last)` is lexicographically sorted with respect to `operator<` or `comp`. If the previous partial permutation does not exist, it sorts the entire range in reverse order and then applies `std::reverse(middle, last)`.

15 **Returns:** `true` if the previous partial permutation exists, `false` otherwise.

16 **Requires:** The type of `*first` shall satisfy the `Swappable` requirements (37). The range `[middle,last)` shall be sorted in ascending order.

17 **Remarks:** Upon returning false, `[first,middle)` is back to the largest partial permutation, that is, the prefix of the descendingly sorted range, and the requirements met for another application of `prev_partial_permutation`.

18 **Complexity:** At most `(last - first)` comparisons and `(last - first)` swaps.

19 [*Note:* In order to prepare the range `[first,last)` for an enumeration of all partial permutations in reverse lexicographic order, `sort(first,last)` or `sort(first,last,comp)` and then `std::reverse(first,last)` followed by `std::reverse(middle,last)`. — *end Note*]

```
template <class BidirectionalIterator>
  bool next_combination(BidirectionalIterator first,
                        BidirectionalIterator middle,
                        BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
  bool next_combination(BidirectionalIterator first,
                        BidirectionalIterator middle,
                        BidirectionalIterator last, Compare comp);
```

20 **Effects:** `next_combination` takes a sequence defined by the range `[first,last)` such that `[first,middle)` stores a *combination*, i.e., some sorted subsequence of `[first,last)`, and permutes it such that `[first,middle)` stores the next combination of the same size from `[first,last)`, and `[middle,last)` is sorted. The next combination is found by assuming that the set of all combinations of a given size from `[first,last)` is lexicographically sorted with respect to `operator<` or `comp`. If the next combination does not exist, it transforms `[first,middle)` into the smallest combination, leaving the entire range sorted.

21 **Returns:** `true` if the next combination exists, `false` otherwise.

22 **Requires:** The type of `*first` shall satisfy the `Swappable` requirements (37). The ranges `[first,middle)` and `[middle,last)` shall both be sorted in ascending order.

23 **Remarks:** Upon returning false, `[first,middle)` is back to the smallest combination, that is, the prefix of the ascendingly sorted range, and the requirements met for another application of `next_combination`.

24 **Complexity:** At most `(last - first)` comparisons and `(last - first)` swaps.

25 [*Note:* In order to prepare the range `[first,last)` for an enumeration of all combinations in lexicographic order, `std::sort(first,last)` or `std::sort(first,last,comp)`. — *end Note*]

```
template <class BidirectionalIterator>
  bool
  prev_combination(BidirectionalIterator first,
                   BidirectionalIterator middle,
                   BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
  bool
  prev_combination(BidirectionalIterator first,
                   BidirectionalIterator middle,
                   BidirectionalIterator last, Compare comp);
```

26 **Effects:** `prev_combination` takes a sequence defined by the range `[first,last)` such that `[first,middle)` stores a *combination*, i.e., some sorted subsequence of `[first,last)`, and permutes it such that `[first,middle)` stores the previous combination of the same size from `[first,last)`, and `[middle,last)` is sorted. The previous combination is found by assuming that the set of all combinations of a given size from `[first,last)` is lexicographically sorted with respect to `operator<` or `comp`. If the previous combination does not exist, it transforms `[first,middle)` into the largest combination, leaving `[middle,last)` sorted.

27 **Returns:** `true` if the previous combination exists, `false` otherwise.

28 **Requires:** The type of `*first` shall satisfy the `Swappable` requirements (37). The ranges `[first,middle)` and `[middle,last)` shall both be sorted in ascending order.

29 **Remarks:** Upon returning false, `[first,middle)` is back to the largest combination, that is, the reversed prefix of the descendingly sorted range, and the requirements met for another application of `prev_combination`.

30 **Complexity:** At most `(last - first)` comparisons and `(last - first)` swaps.

31 [*Note:* In order to prepare the range `[first,last)` for an enumeration of all combinations in reverse lexicographic order, `sort(first,last)` or `sort(first,last,comp)` and then `std::reverse(first,last)` followed by `std::reverse(middle,last)`. — *end Note*]

```
template <class BidirectionalIterator, class T>
```

```
  bool next_mapping(BidirectionalIterator first,
                    BidirectionalIterator last,
                    T first_value, T last_value);

template <class BidirectionalIterator, class T, class Incrementor>
  bool next_mapping(BidirectionalIterator first,
                    BidirectionalIterator last,
                    T first_value, T last_value, Incrementor increment);
```

32 **Effects:** `next_mapping` takes a *mapping*, i.e., a range `[first,last)` from which each value belongs to the range `[first_value,last_value)`, and transforms this sequence into the next mapping from `[first,last)` onto `[first_value,last_value)`. The next mapping is found by assuming that the set of all mappings is lexicographically sorted with respect to the values in the range `[first_value,last_value)`. If such a mapping does not exist, it transforms the mapping into the lexicographically smallest mapping, that is, each value in `[first,last)` equals `first_value`.

33 **Returns:** `true` if the next mapping exists, `false` otherwise.

34 **Requires:** `T` shall meet the requirements of `CopyConstructible` (34) and `Assignable` (23.1) types. In the first form, `T` shall have an `operator++`; in the second form, `Incrementor` shall meet the requirements of `CopyConstructible` (34) types; `last_value` shall be reachable from `first_value` by a finite positive number of evaluations of `++first_value` or `increment(first_value)`. [*Note:* `T` is not required to be LESSTHANCOMPARABLE, instead the order is induced by the increment operator, with each value `x` less than `++x`. — *end Note*]

35 **Complexity:** At most `(last - first)` decrements of `BidirectionalIterator` and `(last - first)` increments of `T`.

```
template <class BidirectionalIterator, class T>
  bool prev_mapping(BidirectionalIterator first,
                    BidirectionalIterator last,
                    T first_value, T last_value);

template <class BidirectionalIterator, class T, class Incrementor>
  bool prev_mapping(BidirectionalIterator first,
                    BidirectionalIterator last,
                    T first_value, T last_value, Incrementor increment);
```

36 **Effects:** `prev_mapping` takes a *mapping*, i.e., a sequence defined by the range `[first,last)` where each value in this range belongs to the range `[first_value,last_value)`, and transforms this sequence into the previous mapping from `[first,last)` onto `[first_value,last_value)`. The previous mapping is found by assuming that the set of all mappings is lexicographically sorted with respect to the values in the range `[first_value,last_value)`. If such a mapping does not exist, it transforms the mapping into the lexicographically smallest mapping, that is, each value in `[first,last)` equals `first_value`.

37 **Returns:** `true` if the previous mapping exists, `false` otherwise.

38 **Requires:** `T` shall meet the requirements of `CopyConstructible` (34) and `Assignable` (23.1) types. In the first form, `T` shall have an `operator--`; in the second form, `Decrementor` shall meet the requirements of `CopyConstructible` (34) types; `first_value` shall be reachable from `last_value` by a finite positive number of evaluations of `--last_value` or `decrement(last_value)`. [*Note:* `T` is not required to be LESSTHANCOMPARABLE, instead the order is induced by the increment operator, with each value `x` less than `++x`. — *end Note*]

39 **Complexity:** At most `(last - first)` decrements of `BidirectionalIterator` and `(last - first)` decrements of `T`.

```
template <class BidirectionalIterator>
  bool next_repeat_combination_counts(BidirectionalIterator first,
                                      BidirectionalIterator last);
```

40 **Effects:** `next_repeat_combination_counts` takes a sequence defined by the range `[first,last)` where each value in this range is a *combination count* and the sum of all counts in this range equals some `total_size` (or `0` if `[first,last)` is empty), and transforms this sequence into the next combination counts of the same `total_size`. The next combination counts are found by assuming that the set of all combination counts whose sums equal `total_size` is lexicographically sorted with respect to the values in the range `[first,last)`. If such a mapping does not exist, it transforms the combination counts into the lexicographically smallest combination counts, that is, `0` for each iterator in `[first,last)` except `total_size` for the last such iterator (if any).

41 **Returns:** `true` if the next combination counts exist, `false` otherwise.

42 **Requires:** The type of `*first` shall satisfy the `Swappable` requirements (37), and be a numeric type (26.1) which can be incremented and decremented. [*Note:* This type is *not* required to have an `operator+` to compute `total_size`. The invariance of the `total_size` mentioned in the effects rule is achieved by applying an equal number of increments and decrements. — *end Note*]

43 **Complexity:** At most `(last - first)` decrements of `BidirectionalIterator` and one increment and one decrement of the type of `*first`.

44 [*Note:* The underlying combinations with repetitions computed from the combination counts are not enumerated in lexicographic order. — *end Note*]

```
template <class BidirectionalIterator>
  bool prev_repeat_combination_counts(BidirectionalIterator first,
                                      BidirectionalIterator last);
```

45 **Effects:** `prev_repeat_combination_counts` takes a sequence defined by the range `[first,last)` where each value in this range is a *combination count* and the sum of all counts in this range equals some `total_size` (or `0` if `[first,last)` is empty), and transforms this sequence into the previous combination counts of the same `total_size`. The previous combination counts are found by assuming that the set of all combination counts whose sums equal `total_size` is lexicographically sorted with

respect to the values in the range `[first,last)`. If such a mapping does not exist, it transforms the combination counts into the lexicographically largest combination counts, that is, `0` for each iterator in `[first,last)` except `total_size` for `first` (if different from `last`).

46 **Returns:** `true` if the previous combination counts exist, `false` otherwise.

47 **Requires:** The type of `*first` shall satisfy the `Swappable` requirements (37), and be a numeric type (26.1) which can be incremented and decremented. [*Note:* This type is *not* required to have an `operator+` to compute `total_size`. The invariance of the `total_size` mentioned in the effects rule is achieved by applying an equal number of increments and decrements. — *end Note*]

48 **Complexity:** At most `(last - first)` decrements of `BidirectionalIterator` and one increment and one decrement of the type of `*first`.

49 [*Note:* The underlying combinations with repetitions computed from the combination counts are not enumerated in reverse lexicographic order. — *end Note*]

6 Illustration

We decided to include this section into the proposal, despite its significant length commitment, for the obvious benefits of clarifying the concepts involved.

6.1 Examplifying `next_partial_permutation`

Given a set $\{0,1,\ldots n-1\}$ of size n, we wish to enumerate all the permutations of size $r \leq n$ that have no repetitions. We know before-hand that there are $n!/(n-r)!$ such permutations. Enumerating them is easy using the `next_partial_permutation` algorithm:

```
const int r = 3, n = 6;
std::vector<int> v(n);
for (int i = 0; i < n; ++i) v[i] = i;

int N = 0;
do {
    ++N;
    std::cout << "[␣" << v[0];
    for (int j = 1; j < r; ++j) { std::cout << ",␣" << v[j]; }
    std::cout << "␣␣]␣";
} while (next_partial_permutation(v.begin(), v.begin() + r, v.end()));
std::cout << "\nFound␣" << N << "␣permutations␣of␣size␣3␣without␣repetitions"
          << "␣from␣a␣set␣of␣size␣5." << std::endl;
```

This code will print out (abridged, broken on several lines):

```
[ 0, 1, 2 ] [ 0, 1, 3 ] [ 0, 1, 4 ] [ 0, 1, 5 ] [ 0, 2, 1 ]
[ 0, 2, 3 ] [ 0, 2, 4 ]          .   .   .          [ 5, 3, 2 ]
```

```
[ 5, 3, 4 ] [ 5, 4, 0 ] [ 5, 4, 1 ] [ 5, 4, 2 ] [ 5, 4, 3 ]
Found 120 permutations of size 3 without repetitions from a set of size 5.
```

The situation is a bit more complex to understand when the original sequence has duplicate values, because the notion of permutation without repetitions in this case consists of the subsequences indexed by all collections of *distinct* r indices among the n indices, with identical such subsequences enumerated only once. Suppose for instance that v contains the sequence $\{0,1,2,0,1,3\}$. In order to start the enumeration, the sequence must first be sorted into $\{0,0,1,1,2,3\}$, then all the sequence of distinct three indices (permutations of size r of $\{0,...n-1\}$) are applied to the above sequence, but notice for instance that the first two sequence of indices generate the same permutation $[0,0,1]$ since the original sequence has the same element at indices 2 and 3. This subsequence is enumerated only once. Here is the code (identical, except for the initialization of v):

```
v[0] = 0; v[1] = 1; v[2] = 2; v[3] = 0; v[4] = 1; v[5] = 3;
std::sort(v.begin(), v.end());

N = 0;    // note: r and n are still 3 and 6, respectively
do {
    ++N;
    std::cout << "[␣" << v[0];
    for (int j = 1; j < r; ++j) { std::cout << ",␣" << v[j]; }
    std::cout << "␣␣]␣";
} while (next_partial_permutation(v.begin(), v.begin() + r, v.end()));
std::cout << "\nFound␣" << N << "␣permutations␣of␣size␣3␣without␣repetitions"
          << "␣from␣a␣(multi)set␣of␣size␣5." << std::endl;
```

This time, the code outputs the much shorter (again, broken on several lines):

```
[ 0, 0, 1 ] [ 0, 0, 2 ] [ 0, 0, 3 ] [ 0, 1, 0 ] [ 0, 1, 1 ] [ 0, 1, 2 ]
[ 0, 1, 3 ] [ 0, 2, 0 ] [ 0, 2, 1 ] [ 0, 2, 3 ] [ 0, 3, 0 ] [ 0, 3, 1 ]
[ 0, 3, 2 ] [ 1, 0, 0 ] [ 1, 0, 1 ] [ 1, 0, 2 ] [ 1, 0, 3 ] [ 1, 1, 0 ]
[ 1, 1, 2 ] [ 1, 1, 3 ] [ 1, 2, 0 ] [ 1, 2, 1 ] [ 1, 2, 3 ] [ 1, 3, 0 ]
[ 1, 3, 1 ] [ 1, 3, 2 ] [ 2, 0, 0 ] [ 2, 0, 1 ] [ 2, 0, 3 ] [ 2, 1, 0 ]
[ 2, 1, 1 ] [ 2, 1, 3 ] [ 2, 3, 0 ] [ 2, 3, 1 ] [ 3, 0, 0 ] [ 3, 0, 1 ]
[ 3, 0, 2 ] [ 3, 1, 0 ] [ 3, 1, 1 ] [ 3, 1, 2 ] [ 3, 2, 0 ] [ 3, 2, 1 ]
Found 42 permutations of size 3 without repetitions of a (multi)set of size 5.
```

Note that if we had wanted the permutations that had no repetitions of values (as opposed to repetitions of indices), we could simply have removed the duplicates from v, yielding a short sequence of m values, and enumerated the permutations of r elements taken from these m values without repetitions. Here m would be 4, and there would be four such permutations.

6.2 Examplifying `next_combination`

Given a set $\{0, 1, ...n-1\}$, we wish to enumerate all the subsets of size $r \leq n$. This is easy using the `next_combination` algorithm that takes three or four arguments:

```
const int r = 3, n = 10;
std::vector<int> v_int(n);
for (int i = 0; i < n; ++i) { v_int[i] = i; }

int N = 0;
do {
    ++N;
    std::cout << "[␣" << v_int[0];
    for (int j = 1; j < r; ++j) { std::cout << ",␣" << v_int[j]; }
    std::cout << "␣]␣";
} while (next_combination(v_int.begin(), v_int.begin() + r, v_int.end()));
std::cout << "\nFound␣" << N << "␣combinations␣of␣size␣" << r << "␣without␣repetitio
          << "␣from␣a␣set␣of␣" << n << "␣elements." << std::endl;
```

This code will print out:

```
[ 0, 1, 2 ] [ 0, 1, 3 ] [ 0, 1, 4 ] [ 0, 1, 5 ] [ 0, 1, 6 ]
[ 0, 1, 7 ] [ 0, 1, 8 ] [ 0, 1, 9 ] [ 0, 2, 3 ]   .  .  .
[ 5, 8, 9 ] [ 6, 7, 8 ] [ 6, 7, 9 ] [ 6, 8, 9 ] [ 7, 8, 9 ]
Found 120 combinations of size 3 without repetitions from a set of 10 elements.
```

Had some elements been equal, e.g., `v = 0, 1, 2, 3, 1, 2, 3, 1, 2, 3`, we would have had the following output instead (the discussion is the same as for permutations without repetitions, except that the sequences in the output must be sorted):

```
[ 0, 1, 1 ] [ 0, 1, 2 ] [ 0, 1, 3 ] [ 0, 2, 2 ] [ 0, 2, 3 ] [ 0, 3, 3 ]
[ 1, 1, 1 ] [ 1, 1, 2 ] [ 1, 1, 3 ] [ 1, 2, 2 ] [ 1, 2, 3 ] [ 1, 3, 3 ]
[ 2, 2, 2 ] [ 2, 2, 3 ] [ 2, 3, 3 ] [ 3, 3, 3 ]
Found 16 combinations of size 3 without repetitions from a (multi)set of 10 elements
```

Suppose we have a slightly different requirement, with a type that cannot be swapped, either because it isn't efficient, or because the sequence cannot be permuted (if it is passed const, for instance). For expository purposes, we use a non-modifiable vector of `std::string`. In this case we can apply the previous solution with an extra level of indirection, and apply `next_combination` to a vector that holds iterators into our non-modifiable vector of strings (note that the iterators are in sorted order):

```
const char *strings[] = {
    "one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "ten"
};
const int m = sizeof strings / sizeof *strings;
std::vector<std::string> v_strings(strings, strings + m);

std::vector<std::vector<std::string>::const_iterator> w(m);
for (int i = 0; i < m; ++i) { w[i] = v_strings.begin() + i; }
```

```
N = 0;    // note: r and n are still 3 and 10, respectively
do {
    ++N;
    std::cout << "[␣" << *w[0];
    for (int j = 1; j < r; ++j) { std::cout << ",␣" << *w[j]; }
    std::cout << "␣␣]" << std::endl;
} while (next_combination(w.begin(), w.begin() + r, w.end()));
std::cout << "\nFound␣" << N << "␣combinations␣of␣size␣" << r << "␣without␣repetitio
          << "␣from␣a␣set␣of␣" << n << "␣elements." << std::endl;
```

This prints out, while never modifying the value of v:

```
[ one, two, three ] [ one, two, four ] [ one, two, four ] [ one, two, five ]
[ one, two, seven ] [ one, two, eight ] [ one, two, nine ] [ one, two, ten ]
[ one, three, four ]   .  .  .  [ seven, eight, nine ] [ seven, eight, ten ]
[ seven, nine, ten ] [ eight, nine, ten ]
Found 120 combinations of size 3 without repetitions from a set of 10 elements.
```

6.3 Examplifying `next_mapping`

Given a set $\{0, 1, ...n-1\}$ of size n, we wish to enumerate all the permutations of size r that possibly have repetitions. (Note that r and n can be in any relation, larger or smaller.) We know before-hand that there are n^r such permutations. Each permutation with repetition is simply an assignment of r variables `x[0] . . . x[r]` into the set $\{0, 1, ...n-1\}$. Enumerating them is easy using the `next_mapping` algorithm:

```
const int r = 5, n = 3;
std::vector<int> v_int(r, 0);

int N = 0;
do {
    ++N;
    std::cout << "[␣" << v_int[0];
    for (int j = 1; j < r; ++j) { std::cout << ",␣" << v_int[j]; }
    std::cout << "␣]␣";
} while (next_mapping(v_int.begin(), v_int.end(), 0, n));
std::cout << "\nFound␣" << N << "␣mappings␣from␣" << n
          << "␣positions␣to␣a␣set␣of␣" << r << "␣elements." << std::endl;
```

This code will print out:

```
[ 0, 0, 0, 0, 0 ] [ 0, 0, 0, 0, 1 ] [ 0, 0, 0, 0, 2 ] [ 0, 0, 0, 1, 0 ]
[ 0, 0, 0, 1, 1 ] [ 0, 0, 0, 1, 2 ] [ 0, 0, 0, 2, 0 ] [ 0, 0, 0, 2, 1 ]
[ 0, 0, 0, 2, 2 ] [ 0, 0, 1, 0, 0 ] [ 0, 0, 1, 0, 1 ] [ 0, 0, 1, 0, 2 ]
  .  .  .  .  .                     .  .  .  .  .   [ 2, 2, 2, 1, 1 ]
[ 2, 2, 2, 1, 2 ] [ 2, 2, 2, 2, 0 ] [ 2, 2, 2, 2, 1 ] [ 2, 2, 2, 2, 2 ]
Found 243 mappings from 3 positions to a set of 5 elements.
```

Note that this exactly enumerates the coordinates of a point in a r-dimensional cubical grid whose side has length n. Also note that there isn't any special treatment for the range values, they are simply consecutive values in the sense that one goes from one value to the next by using `operator++`. Note that `first_value` is not necessarily an iterator, because there is no requirement on a dereferencing `operator*`. However, an iterator can be used. Note that the values pointed to by these iterators, or their orderings, are irrelevant. We demonstrate this here using instead of the range `[0, r)` a range `[w.begin(), w.end())` into some vector of fruits.

```
const char *strings[] = { "banana", "peach", "apple" };
std::vector<std::string> W(strings, strings + n);
const std::vector<std::string>& w = W;
std::vector<std::vector<std::string>::const_iterator> v_iter(r, w.begin());

N = 0;    // note: r and n are still 5 and 3, respectively
do {
    ++N;
    std::cout << "[␣" << *v_iter[0];
    for (int j = 1; j < r; ++j) { std::cout << ",␣" << *v_iter[j]; }
    std::cout << "␣␣]" << std::endl;
} while (next_mapping(v_iter.begin(), v_iter.end(), w.begin(), w.end()));
std::cout << "\nFound␣" << N << "␣mappings␣from␣" << n
          << "␣positions␣to␣a␣set␣of␣" << r << "␣fruits." << std::endl;
```

This code then outputs:

```
[ banana, banana, banana, banana, banana ] [ banana, banana, banana, banana, peach ]
[ banana, banana, banana, banana, apple ] [ banana, banana, banana, peach, banana ]
[ banana, banana, banana, peach, peach ] [ banana, banana, banana, peach, apple ]
[ banana, banana, banana, apple, banana ] [ banana, banana, banana, apple, peach ]
[ banana, banana, banana, apple, apple ] [ banana, banana, peach, banana, banana ]
[ banana, banana, peach, banana, peach ] [ banana, banana, peach, banana, apple ]
                . . . . .                [ apple, apple, apple, peach, peach ]
[ apple, apple, apple, peach, apple ] [ apple, apple, apple, apple, banana ]
[ apple, apple, apple, apple, peach ] [ apple, apple, apple, apple, apple ]
Found 243 mappings from 5 positions to a set of 3 fruits.
```

6.4 Examplifying `next_combination_counts`

Given a set $\{0,1,...n-1\}$, we wish to enumerate all the combinations of size r of elements in the set, taken with repetitions. This is easy using the `next_combination_counts` algorithm that takes only two arguments. Instead of copying (possibly multiple repetitions of) elements into a range, we simply require a range holding the multiplicities of each instance in the original set. The lexicographically first assignment of multiplicities with a total multiplicity of size r is $\{0,...0,r\}$.

```
const int r = 5, n = 3;
std::vector<int> multiplicities(n, 0);
```

```
multiplicities.back() = r;

int N = 0;
do {
    ++N;
    std::cout << "[␣" << multiplicities[0];
    for (int j = 1; j < n; ++j) { std::cout << ",␣" << multiplicities[j]; }
    std::cout << "␣␣]" << std::endl;
} while (next_combination_counts(multiplicities.begin(), multiplicities.end()));
std::cout << "Found␣" << N << "␣combinations␣of␣size␣" << r
          << "␣with␣repetitions␣from␣a␣set␣of␣" << n << "␣elements." << std::endl;
```

This code will print out:

```
[ 0, 0, 5 ] [ 0, 1, 4 ] [ 0, 2, 3 ] [ 0, 3, 2 ] [ 0, 4, 0 ] [ 0, 5, 0 ]
[ 1, 0, 4 ] [ 1, 1, 3 ] [ 1, 2, 2 ] [ 1, 3, 1 ] [ 1, 4, 0 ] [ 2, 0, 3 ]
[ 2, 1, 2 ] [ 2, 2, 1 ] [ 2, 3, 0 ] [ 3, 0, 2 ] [ 3, 1, 1 ] [ 3, 2, 0 ]
[ 4, 0, 1 ] [ 4, 1, 0 ] [ 5, 0, 0 ]
Found 21 combinations of size 5 with repetitions from a set of 3 elements.
```

Note that it isn't hard to actually display the sequence itself. Given a value of multiplicities, e.g., $\{1,3,1\}$, we can copy into a container w the combination with repetition of a vector v.

```
const char *strings[] = { "banana", "peach", "apple" };
std::vector<std::string> v(strings, strings + n);

multiplicities[0] = 1;  multiplicities[1] = 3;  multiplicities[2] = 1;
assert(r == multiplicities[0] + multiplicities[1] + multiplicities[2]);

std::vector<std::string> w;
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < multiplicities[i]; ++j) {
        w.push_back(v[i]);
    }
}
assert(r == w.size());
```

Printing the container holding the repeated values produces:

```
[ banana, peach, peach, peach, apple ]
```

7 Reference implementation

We offer a reference implementation to show both that the implementation is not overly difficult, but that it is also far from trivial. We limit ourselves to the non-predicated version. The first implementation is a gem of simplicity, building on `std::next_permutation`:

```
template <class BidirectionalIterator>
  bool next_partial_permutation(BidirectionalIterator first,
                                BidirectionalIterator middle,
                                BidirectionalIterator last)
{
  reverse (middle, last);
  return next_permutation(first, last);
}

template<class BidirectionalIterator>
  bool prev_partial_permutation(BidirectionalIterator first,
                                BidirectionalIterator middle,
                                BidirectionalIterator last)
{
  bool result = prev_permutation(first, last);
  reverse (middle, last);
  return result;
}
```

It is especially nice to standardize such tricks as many developers will likely come up with a much more complicated (and slower) solution.

The next one is the hardest to get, and emphasizes the value of standardizing these components, as this code is probably too hard or costly to write, even for seasoned developers:

```
namespace {
template<class BidirectionalIterator>
  bool next_combination(BidirectionalIterator first1,
                        BidirectionalIterator last1,
                        BidirectionalIterator first2,
                        BidirectionalIterator last2)
  {
      if ((first1 == last1) || (first2 == last2)) {
          return false;
      }

      BidirectionalIterator m1 = last1;
      BidirectionalIterator m2 = last2; --m2;

      while (--m1 != first1 && !(*m1 < *m2)) {
      }

      bool result = (m1 == first1) && !(*first1 < *m2);

      if (!result) {
          while (first2 != m2 && !(*m1 < *first2)) {
              ++first2;
          }
```

```
            first1 = m1;
            std::iter_swap (first1, first2);
            ++first1;
            ++first2;
        }

        if ((first1 != last1) && (first2 != last2)) {
            m1 = last1;  m2 = first2;
            while ((m1 != first1) && (m2 != last2)) {
                std::iter_swap (--m1, m2);
                ++m2;
            }

            std::reverse (first1, m1);
            std::reverse (first1, last1);

            std::reverse (m2, last2);
            std::reverse (first2, last2);
        }

        return !result;
    }

} // namespace

template<class BidirectionalIterator>
  bool next_combination(BidirectionalIterator first,
                        BidirectionalIterator middle,
                        BidirectionalIterator last)
  {
    return detail::next_combination(first, middle, middle, last);
  }

template<class BidirectionalIterator>
  inline
  bool prev_combination(BidirectionalIterator first,
                        BidirectionalIterator middle,
                        BidirectionalIterator last)
  {
    return detail::next_combination(middle, last, first, middle);
  }
```

The following is equivalent to n identical nested loops, but with a dynamic (run-time) value for n. The algorithm is somewhat simple, and used to enumerate multi-dimensional indices in hypercubes:

```
template <class BidirectionalIterator, class T>
  bool
  next_mapping(BidirectionalIterator first,
               BidirectionalIterator last,
```

```
                T first_value, T last_value)
{
    if (last == first ) {
        return false;
    }
    do {
        if (++(*(--last)) != last_value) {
            return true;
        }
        *last = first_value;
    } while (last != first);
    return false;
}

template <class BidirectionalIterator, class T>
  bool
  prev_mapping(BidirectionalIterator first,
               BidirectionalIterator last,
               T first_value, T last_value)
{
    if (last == first) {
        return false;
    }
    --last_value;
    do {
        if (*(--last) != first_value) {
            --(*last);
            return true;
        }
        *last = last_value;
    } while (last != first);
    return true;
}
```

The last one is not too hard to obtain, at least forward. Backward is a little bit more difficult. Clearly, again, deriving these correctly requires some effort, and it would be better to have those algorithms standardized.

```
template <class BidirectionalIterator>
  bool
  next_repeat_combination_counts(BidirectionalIterator first,
                                 BidirectionalIterator last)
{
    BidirectionalIterator current = last;
    while (current != first && *(--current) == 0) {
    }
    if (current == first) {
        if (first != last && *first != 0)
            std::iter_swap(--last, first);
        return false;
```

```
    }
    --(*current);
    std::iter_swap(--last, current);
    ++(*(--current));
    return true;
}

template <class BidirectionalIterator>
bool
prev_repeat_combination_counts(BidirectionalIterator first,
                               BidirectionalIterator last)
{
    if (first == last)
        return false;
    BidirectionalIterator current = --last;
    while (current != first && *(--current) == 0) {
    }
    if (current == last || current == first && *current == 0) {
        if (first != last)
            std::iter_swap(first, last);
        return false;
    }
    --(*current);
    ++current;
    if (0 != *last) {
        std::iter_swap(current, last);
    }
    ++(*current);
    return true;
}
```

8 Acknowledgements

Thanks to Phil Garofalo, for starting the topic on the Boost mailing list in 2002, to Ben Bear for reviving the topic in November 2007 with his Gacap library, and for helping to test our implementation, to Howard Hinnant for his early contributions and to Jens Seidel for an earlier reading of this proposal.

References

[1] Knuth, D.E. *The Art of Computer Programming.* Volume 4, Fascicle 3, "Generating All Combinations and Partitions", Addison-Wesley Professional, 2005. ISBN 0-201-85394-9.

[2] T. Cormen, C. Leiserson, R. Rivest and C. Stein. *Introduction to algorithms (2nd edition)*. The MIT Press, 2004.

[3] Wikipedia. *Combinatorics*. `http://en.wikipedia.org/wiki/Combinatorics`

[4] Howard Hinnant. Combinations and Permutations. `http://home.twcny.rr.com/hinnant/cpp_extensions/combinations.html`

[5] Hervé Brönnimann. Proposition for Boost.permutation library (in preparation). `http://photon.poly.edu/~hbr/boost/combinations.html`