Set-Theoretic Operations on Virtual Containers: Performing Set Operations On-the-Fly

Gregory Begelman gbegelman@hotmail.com
Lev Finkelstein lev@cs.technion.ac.il
Evgeniy Gabrilovich† gabr@acm.org

Abstract

The Standard Template Library (STL) [1] provides implementation for basic set operations on sorted ranges. When the result of a set operation is only needed temporarily, conventional STL usage schemes require artificial auxiliary objects and lead to clumsy programming style. We propose a more elegant solution, in which the resulting range is built implicitly, so that traversing it incurs no additional overhead. This is achieved by encapsulating the logic of set operations, so that the next element in the output range is computed "on-the-fly". The new template-based implementation of set operations also satisfies all the assumptions that hold for their STL counterparts.

Motivation

The Standard Template Library [1] offers set-theoretic operations *union*, *intersection*, *difference* and *symmetric difference*, accessible through the header file <algorithm>. These operations (or *algorithms* in STL-ese) are applicable to sets, as well as to other sorted ranges with input iterators defined. Given two ranges of input iterators and an output iterator, set algorithms construct a new range implementing the desired operation[1].

When the result of a set operation is only needed for some interim manipulation (e.g., for sweeping over its elements once), two usage scenarios are possible with the conventional STL approach:

1. The outcome range is actually constructed in a temporary data structure, filled through an *insert iterator*[2]. Listing 1 shows an example of this approach that computes an intersection of two sets of integers.

   Listing 1. Set intersection example using a temporary data structure

```
void temp_intersection() {
  vector<int> vec1, vec2, temp_result;
  … // prepare input sequences in 'vec1' and 'vec2'
  set_intersection(vec1.begin(), vec1.end(), vec2.begin(), vec2.end(),
                   back_inserter(temp_result)); // build the result in a temporary container

  for (int i = 0; i < temp_result.size(); ++i) {
      // do something
```

† Corresponding author (email: gabr@acm.org).

[1] In what follows, we also consider the *merge* algorithm, which is very similar conceptually to the above four, although it is not a set-theoretic operation *per se*.

[2] Insert iterator is an adaptor which implements the interface of output iterator, so that any element assigned to it is inserted into the underlying container; see paragraph 24.4.2 ([lib.insert.iterators]) in [1].

```
  }
}
```

An obvious drawback of this approach is the overhead to create (and eventually to deallocate) this auxiliary structure with all the element copying involved.

2. The code fragment to be applied to the outcome range is encapsulated in an iterator-like object that satisfies all the assumptions of an output iterator. This object is passed as an output parameter to set operations, and thus gets invoked on each element of the resulting range like a callback function. Sample implementation of this technique for the intersection operation is given in Listing 2.

   Listing 2. Set intersection example using a callback function

```
class callback {
        void do_something(int v) {
                // do something
        }
public:
        callback &operator=(int v) { do_something(v); return *this;}
        callback &operator*() { return *this; }
        // op++ for output iterators has a dummy implementation
        callback &operator++() { return *this; }
        callback &operator++(int) { return *this; }
};

void callback_intersection() {
  vector<int> vec1, vec2;
  …
  set_intersection(vec1.begin(), vec1.end(), vec2.begin(), vec2.end(), callback());
}
```

   A disadvantage of this technique is that detaching the code from regular program flow and encapsulating it in an auxiliary object makes the implementation quite cumbersome (especially when the code fragment uses multiple external variables defined elsewhere).

We propose a solution (see Listing 3 below) in which the output range is built in a "lazy" manner, its elements being only computed when needed. To this end, we define a *virtual*[3] container featuring a (constant) input iterator. Successively incrementing this iterator virtually traverses the elements of the resulting range in the correct order, *without actually constructing the range*. Note that the iterator has to be constant, since otherwise modifying the elements it points to might invalidate the ordering of input ranges.

[3] The use of word "virtual" here has nothing to do with virtual functions.

Listing 3. New ("on-the-fly") implementation of set intersection

```
void online_intersection() {
  vector<int> vec1, vec2;
  …
  typedef set_intersection_online<int, vector<int>::iterator>  IntersectionOnline;
  IntersectionOnline res(vec1.begin(), vec1.end(), vec2.begin(), vec2.end());

  for (IntersectionOnline::const_iterator iter = res.begin(); iter != res.end(); ++iter) {
        // do something
  }
}
```

The proposed solution provides template-based classes (virtual containers) set_union_online, set_intersection_online, set_difference_online, set_symmetric_difference_online, and merge_online[4]. The difference in logic of these operations is realized in different implementations of the increment and the dereference operator of each container's iterator (op++() and op*(), respectively).

Apparently, the approaches of both Listing 2 and Listing 3 encapsulate the algorithm logic, but the latter has an advantage of doing this only once. While the callback function needs to be specifically designed for each application, we wrap the algorithms of the various set operations in smart iterators that can later be used elsewhere without any need for adaptation.

Implementation

In this section we evolve a "lazy" implementation of set operations. We first describe the structure of a (constant) input iterator, which is available in all virtual containers for traversing their elements. We then show the base class for the virtual containers. Finally, we describe the implementation of the different containers *per se*.

Smart iterator

Let us review the sample scenario of Listing 3. In order to perform a particular set operation, we first instantiate the appropriate virtual container, and then use its iterator[5] to run through the elements of the output range. To facilitate this approach, the algorithm for computing the desired operation (e.g., set_intersection) needs to be encapsulated in the iterator itself. The iterator's increment and dereference operators are thus defined in terms of three auxiliary functions – _pre_increment()[6], _dereference() and _find_next() (the latter being used to identify the next element in the output range) – which actually realize the specific algorithm in hand. The iterator template shown in Listing 4 only *declares*

[4] Observe that the inplace_merge algorithm is not suitable for this treatment, as its main purpose is to *actually* merge its input ranges, doing so in the same memory space where the original elements reside (in-place).

[5] Note that we use here an *input* iterator, so that dereferencing it yields the desired elements *as if the operation outcome range has actually been constructed.*

[6] In order to prevent code duplication, we only implement the *pre*-increment operator, which is then used to implement the *post*-increment version (see Listing 4).

these auxiliary functions, while their full implementation is given later for each particular algorithm (set_union_online, set_intersection_online etc.).

Implementation note: As we strived to make the code compatible with a number of major C++ compilers, and since Microsoft Visual C++ does not support partial template specialization, we opted here for a slightly bulky, but portable approach. The iterator template declares the three auxiliary functions *for all available algorithms*, but later each algorithm only *implements* those functions pertinent to its iterator proper. The linker eventually gets rid of all the "unused" declarations. This scheme operates a variant of tag dispatching mechanism [2], using the definitions of tags and supplementary macros of Listing 5.

The parameters of the iterator class template are as follows:

- class _T – value type
- class _Iter1, class _Iter2 – iterator types for the two input ranges
- class _StrictWeakOrdering – a binary predicate for comparing the objects of the two input ranges
- class _Tag – an auxiliary class for distinguishing the implementations of various set operations through the tag dispatching mechanism

Listing 4. Smart iterator

```
template<class _T, class _Iter1, class _Iter2, class _StrictWeakOrdering, class _Tag>
class _const_set_online_iterator {
public:
  typedef _T value_type;
private:
  // Iterators to the current and last elements of the input ranges
  _Iter1 _current1, _last1;
  _Iter2 _current2, _last2;

  // Possible outcomes of comparing the current elements in the input ranges:
  // FIRST     : _StrictWeakOrdering(*current1, *current2) == true
  //               or the second range has been exhausted
  // SECOND : _StrictWeakOrdering(*current2, *current1) == true
  //               or the first range has been exhausted
  // EQUAL    : neither the first nor the second condition is true
  enum compare_state { FIRST, SECOND, EQUAL };

  // The comparison state is used in iterator increment and dereference
  compare_state _compare;

  compare_state get_compare_state() const {
          if (_current1 == _last1) return SECOND;
          if (_current2 == _last2) return FIRST;
          if (_StrictWeakOrdering()(*_current1, *_current2)) return FIRST;
```

```
            if (_StrictWeakOrdering()(*_current2, *_current1)) return SECOND;
            return EQUAL;
    }

    // The following functions need to be defined with the corresponding tag parameter:
    // inline _const_set_online_iterator& _pre_increment(const _Tag&);
    // inline void _find_next(const _Tag&);
    // inline const value_type _dereference(const _Tag&) const;
    //
    // Since MS VC++ des not support partial template specialization we declare these functions
    // for all possible tags. The implementation, however, is provided only for the pertinent tag.
    INSTANTIATE_FOR_ALL_OPERATIONS(DECLARE_PRE_INCREMENT)
    INSTANTIATE_FOR_ALL_OPERATIONS(DECLARE_DEREFERENCE)
    INSTANTIATE_FOR_ALL_OPERATIONS(DECLARE_FIND_NEXT)

    void find_next() {
            _find_next(_Tag());
    }

public:
    // The constructor takes two boundary elements for each of the input ranges
    _const_set_online_iterator(const _Iter1& current1, const _Iter1& last1,
                                    const _Iter2& current2, const _Iter2& last2) :
            _current1(current1), _last1(last1), _current2(current2), _last2(last2)
    {
            _compare = get_compare_state();  // update compare state for the current elements

            find_next();  // find the next element for which the predicate is true
    }

    // Basic operators
    bool operator==(const _const_set_online_iterator& rhs) const {
            return ((_current1 == rhs._current1) && (_current2 == rhs._current2));
    }

    bool operator!=(const _const_set_online_iterator& rhs) const {
            return !operator==(rhs);
    }

    _const_set_online_iterator& operator++() {
            return _pre_increment(_Tag());
    }

    _const_set_online_iterator& operator++(int) {
            _const_set_online_iterator temp = *this;
            ++(*this);
            return temp;
```

```
  }

  const value_type operator*() const {
          return _dereference(_Tag());
  }
};
```

Listing 5. Definitions for the tag dispatching mechanism

```
struct union_tag {};
struct intersection_tag {};
struct difference_tag {};
struct symmetric_difference_tag {};
struct merge_tag {};

// Declaration of the pre-increment operator
#define DECLARE_PRE_INCREMENT(tag)                      \
inline _const_set_online_iterator& _pre_increment(const tag&);

// Declaration of the dereference operator
#define DECLARE_DEREFERENCE(tag)                        \
inline const value_type &_dereference(const tag&) const;

// Declaration of 'find_next' function
#define DECLARE_FIND_NEXT(tag)                          \
inline void _find_next(const tag&);

#define INSTANTIATE_FOR_ALL_OPERATIONS(MACRO)           \
MACRO(union_tag)                                        \
MACRO(intersection_tag)                                 \
MACRO(difference_tag)                                   \
MACRO(symmetric_difference_tag)                         \
MACRO(merge_tag)
```

Base class for "virtual containers"

All virtual containers implementing various set operations derive from the base class shown in Listing 6. The template parameters are mainly used to instantiate the smart iterator, and are therefore identical to those of Listing 4. The base class contains the functionality shared by all set operations, namely, the iterator access functions begin() and end().

Listing 6. Base class for "virtual containers"

```
template<class _T, class _Iter1, class _Iter2, class _StrictWeakOrdering, class _Tag>
class _set_online {
  _Iter1 _first1, _last1;
```

```
  _Iter2 _first2, _last2;

public:
  typedef _T value_type;
  typedef _const_set_online_iterator<_T, _Iter1, _Iter2, _StrictWeakOrdering, _Tag>
          _const_iterator;

  _set_online(_Iter1 first1, _Iter1 last1, _Iter2 first2, _Iter2 last2) :
          _first1(first1), _last1(last1), _first2(first2), _last2(last2) {}

  _const_iterator begin() const { return _const_iterator(_first1, _last1, _first2, _last2); }

  _const_iterator end() const { return _const_iterator(_last1, _last1, _last2, _last2); }
};
```

Set algorithms

This section outlines the core implementation of set algorithms, using set intersection operation as an example. The code fragment given in Listing 7 first instantiates the container itself (using the symbolic intersection_tag), and then implements the three auxiliary functions of the corresponding iterator:

- Function _pre_increment(), which underlies the implementation of op++(), advances the current position in both of its input ranges, and then scans them for the next element to be included in the intersection.
- Function _dereference(), used in the dereference operator op*(), simply returns the element found earlier by an increment operator[7] or the iterator constructor (should the dereference be invoked immediately upon the iterator construction).
- Function _find_next() implements the essence of the set intersection algorithm, looking for the next element to be added to the resultant set.

The implementation in Listing 7 uses a set of auxiliary macros to instantiate all the templates involved; these are depicted in Listing 8. Implementation of other algorithms is mostly similar to that of set intersection. The complete code of this article can be obtained from the Dr. Dobb's Journal Web site at http://www.ddj.com/ftp/.

Listing 7. Implementation of the set intersection algorithm (set_intersection_online)

```
INSTANTIATE_SET_ONLINE(set_intersection_online, intersection_tag)

INSTANTIATE_PRE_INCREMENT(intersection_tag)
{
  ++_current1;
  ++_current2;
  find_next();
  return *this;
```

[7] Either pre- or post-increment one.

```
}

INSTANTIATE_DEREFERENCE(intersection_tag)
{
  return *_current1;
}

INSTANTIATE_FIND_NEXT(intersection_tag)
{
  if (_current1 == _last1 || _current2 == _last2) {
            _current1 = _last1;
            _current2 = _last2;
            return;
  }

  while (_StrictWeakOrdering()(*_current1, *_current2) ||
         _StrictWeakOrdering()(*_current2, *_current1))
  {
            while ((_current1 != _last1) &&
                     _StrictWeakOrdering()(*_current1, *_current2))
                     ++_current1;

            if (_current1 == _last1) {
                     _current2 = _last2;
                     return;
            }

            while ((_current2 != _last2) &&
                     _StrictWeakOrdering()(*_current2, *_current1))
                     ++_current2;

            if (_current2 == _last2) {
                     _current1 = _last1;
                     return;
            }
  }
}
```

Listing 8. Auxiliary macros for instantiating individual virtual containers

```
// Auxiliary macro for instantiating "virtual containers"
#define INSTANTIATE_SET_ONLINE(classname, tag)                  \
template<class _T,                                              \
        class _Iter1,                                           \
        class _Iter2 = _Iter1,                                  \
        class _StrictWeakOrdering = std::less<_T> >             \
class classname :                                               \
```

```
  public _set_online<_T, _Iter1, _Iter2, _StrictWeakOrdering, tag>          \
{                                                                           \
public:                                                                     \
  typedef _T value_type;                                                    \
  classname (_Iter1 first1, _Iter1 last1, _Iter2 first2, _Iter2 last2) :    \
          _set_online<_T, _Iter1, _Iter2, _StrictWeakOrdering, tag>         \
  (first1, last1, first2, last2)                                            \
  {}                                                                        \
};

// Auxiliary macro for instantiating the pre-increment operator
#define INSTANTIATE_PRE_INCREMENT(tag)                                      \
template<class _T, class _Iter1, class _Iter2, class _StrictWeakOrdering,   \
          class _Tag>                                                       \
inline _const_set_online_iterator<_T, _Iter1, _Iter2,                       \
                                  _StrictWeakOrdering, _Tag>&               \
  _const_set_online_iterator<_T, _Iter1, _Iter2,                            \
                             _StrictWeakOrdering, _Tag>::                   \
_pre_increment(const tag&)

// Auxiliary macro for instantiating the dereference operator
#define INSTANTIATE_DEREFERENCE(tag)                                                \
template<class _T, class _Iter1, class _Iter2, class _StrictWeakOrdering,           \
          class _Tag>                                                               \
inline const _const_set_online_iterator<_T, _Iter1, _Iter2,                         \
                                        _StrictWeakOrdering, _Tag>::value_type      \
  &_const_set_online_iterator<_T, _Iter1, _Iter2,                                   \
                              _StrictWeakOrdering, _Tag>::                          \
_dereference(const tag&) const

// Auxiliary macro for instantiating the "find_next" function
#define INSTANTIATE_FIND_NEXT(tag)                                          \
template<class _T, class _Iter1, class _Iter2, class _StrictWeakOrdering,   \
          class _Tag>                                                       \
inline void _const_set_online_iterator<_T, _Iter1, _Iter2,                  \
                                       _StrictWeakOrdering, _Tag>::         \
_find_next(const tag&)
```

Summary

This work suggests a novel implementation of set-theoretic operations in C++, in a way that does not compromise computational efficiency or programming style. This is done by carrying out set operations in a lazy manner, building the output range of elements “on-the-fly”. The proposed implementation satisfies all the assumptions for STL set

operations, namely, time complexity, stability[8] (where applicable), and support for multisets[9].

Interesting additional functionality might be obtained by instantiating the virtual containers developed in this article on *reverse iterators*. When the two input ranges are given by reverse rather than forward iterators, it becomes possible to traverse the outcome of set algorithms *backwards* – a useful extension under certain circumstances.

Future extensions to this work can relax the requirements of uniformity on input and output ranges. For example, in some cases it may be necessary to merge sequences of records with different structure, performing a particular set operation based on (comparable) keys available in both kinds of records. Another extension can generalize over the type of output elements, creating a sequence of elements of some new type, different from that of input items.

Acknowledgments

We are thankful to Alex Gontmakher for his helpful comments.

References

[1] "Information Technology – Programming Languages – C++", International Standard ISO/IEC 14882-1998(E).
[2] "Generic Programming Techniques", C++ Boost. Online document: http://www.boost.org/more/generic_programming.html

About the authors

Gregory Begelman is Algorithm Developer at Zapper Technologies Inc. He holds an M.Sc. degree in Computer Science from the Moscow Institute of Physics and Technology. His interests include computer-assisted simulation modeling and artificial intelligence. He can be reached at gbegelman@hotmail.com.

Lev Finkelstein is System Architect at the Algorithms Group at Zapper Technologies Inc., and is a Ph.D. student in Computer Science at the Technion – Israel Institute of Technology. His interests include artificial intelligence, machine learning, multi-agent systems, and data mining. He can be reached at lev@cs.technion.ac.il.

Evgeniy Gabrilovich is Team Leader of Core Technology at the Algorithms Group at Zapper Technologies Inc. He holds an M.Sc. degree in Computer Science from the Technion – Israel Institute of Technology. His interests involve computational linguistics, information retrieval, artificial intelligence, and speech processing. He can be contacted at gabr@acm.org.

---

[8] The *stability* requirement states that for equivalent elements in the input ranges, the elements from the first range either precede (in case of merge) or supercede (in case of set_union and set_intersection) those from the second. See paragraphs 25.3.4 ([lib.alg.merge]) and 25.3.5 ([lib.alg.set.operations]) in [1].
[9] Multiset is a generalization of set which can contain multiple copies of equivalent elements; see paragraphs 23.3.4 [lib.multiset] and 25.3.5 ([lib.alg.set.operations]) in [1].