Subject: Re: I don't get some aspects of STL ... [pointless rant]
Posted by mr_ped on Thu, 19 Mar 2009 10:13:00 GMT
View Forum Message <> Reply to Message

Here are the "inplace" variants of set_difference for STL, written in a way mimicking the original code as much as possible.

Although I did hit two "major" problems which make these functions a bit different.
1) the original does not need the list container itself, while I need to do erase upon it, so the parameters are now not only from iterator family.
2) the return value points at the start of output range, not end. That's the way how I need it, and I wonder who needs end of range of new result anyway, I mean you need result usually to further use it, and usually the further usage starts to work at the start of result, so I simply can't figure out the reason why STL is returning end, except the simple "STL sucks and defends actively against being used in common programming problems in simple way" reason.
It's either me missing the spirit of STL and trying to abuse it in wrong way, or it's STL completely missing the common programming problems and instead solving unreal scenarios and helping with real ones only by accident and with difficulties.
Rant off, let's talk code .

set_algo.h

```
#ifndef _set_inplace_algo_h_
#define _set_inplace_algo_h_

#include <algorithm>

namespace std
{
  //for license of the original set_difference code see:
  //  .../c++/bits/stl_algo.h file.
  //If you find this different enough to be considered my code,
  //feel free to use it under BSD license. (my point of view)
  //If you feel this is derivative work of original, follow the
  //original license then.

  /**
   *  @brief Modify first range to return the difference of two sorted ranges.
   *  @param  first1  Start of first range.
   *  @param  last1   End of first range.
   *  @param  first2  Start of second range.
   *  @param  last2   End of second range.
   *  @param  list    List containing the first range
   *  @return  Start of modified first range.
   *  @ingroup setoperations
   *
   *  This operation mimicks behavior of set_difference, but the result is not
   *  copy of elements, instead the actual first range is modified (elements
```

```
 *  are erased from first range, if they are found in second range).
 *  The input ranges may not overlap, the output range is always within
 *  the first range.
 *  Return value is start (warning, original set_difference returns end) of
 *  modified first range (i.e. first unerased element or __last1).
*/
template<typename _InputOutputIterator1, typename _InputIterator2,
         typename _ListContainer>
  _InputOutputIterator1
  set_difference_inplace(_InputOutputIterator1 __first1, _InputOutputIterator1 __last1,
                _InputIterator2 __first2, _InputIterator2 __last2,
                _ListContainer & __list)
  {
    // concept requirements
    __glibcxx_function_requires(_OutputIteratorConcept<_InputOutputIterator1,
          typename iterator_traits<_InputOutputIterator1>::value_type>)
    __glibcxx_function_requires(_InputIteratorConcept<_InputIterator2>)
    __glibcxx_function_requires(_SequenceConcept<_ListContainer>)
    __glibcxx_function_requires(_SameTypeConcept<
          typename iterator_traits<_InputOutputIterator1>::value_type,
          typename iterator_traits<_InputIterator2>::value_type>)
    __glibcxx_function_requires(_LessThanComparableConcept<
          typename iterator_traits<_InputOutputIterator1>::value_type>)
    __glibcxx_requires_sorted(__first1, __last1);
    __glibcxx_requires_sorted(__first2, __last2);

    _InputOutputIterator1 __ret = __first1;
    --__ret;      //move to safe place in case __first1 will be erased
    while (__first1 != __last1 && __first2 != __last2)
      if (*__first1 < *__first2)
        ++__first1;
      else if (*__first2 < *__first1)
        ++__first2;
      else
        {
          __first1 = __list.erase( __first1 );
          ++__first2;
        }
    return ++__ret;    //return the first unerased member of list
  }

/**
 *  @brief  Modify first range to return the difference of two sorted
 *  ranges using comparison functor.
 *  @param  first1  Start of first range.
 *  @param  last1   End of first range.
 *  @param  first2  Start of second range.
 *  @param  last2   End of second range.
```

```
 *  @param  comp    The comparison functor.
 *  @param  list    List containing the first range
 *  @return  Start of modified first range.
 *  @ingroup setoperations
 *
 *  This operation mimicks behavior of set_difference, but the result is
 *  not copy of elements, instead the actual first range is modified.
 *  Return value is start (warning, original set_difference returns end) of
 *  modified first range (i.e. first unerased element or __last1).
*/
template<typename _InputOutputIterator1, typename _InputIterator2,
         typename _Compare, typename _ListContainer>
  _InputOutputIterator1
  set_difference_inplace(_InputOutputIterator1 __first1, _InputOutputIterator1 __last1,
                _InputIterator2 __first2, _InputIterator2 __last2,
                _ListContainer & __list, _Compare __comp)
  {
    typedef typename iterator_traits<_InputOutputIterator1>::value_type
      _ValueType1;
    typedef typename iterator_traits<_InputIterator2>::value_type
      _ValueType2;

    // concept requirements
    __glibcxx_function_requires(_OutputIteratorConcept<_InputOutputIterator1,
                          _ValueType1>)
    __glibcxx_function_requires(_InputIteratorConcept<_InputIterator2>)
    __glibcxx_function_requires(_SequenceConcept<_ListContainer>)
    __glibcxx_function_requires(_BinaryPredicateConcept<_Compare,
                          _ValueType1, _ValueType2>)
    __glibcxx_function_requires(_BinaryPredicateConcept<_Compare,
                          _ValueType2, _ValueType1>)
    __glibcxx_requires_sorted_set_pred(__first1, __last1, __first2, __comp);
    __glibcxx_requires_sorted_set_pred(__first2, __last2, __first1, __comp);

    _InputOutputIterator1 __ret = __first1;
    --__ret;      //move to safe place in case __first1 will be erased
    while (__first1 != __last1 && __first2 != __last2)
      if (__comp(*__first1, *__first2))
        ++__first1;
      else if (__comp(*__first2, *__first1))
        ++__first2;
      else
        {
          __first1 = __list.erase( __first1 );
          ++__first2;
        }
    return ++__ret;    //return the first unerased member of list
  }
```

```
} // namespace std

#endif
```

Warning, I didn't test this extensively yet, so there may be some bug hidden. If anyone is up to do a review, I will be really glad for it.
