CB4，Inprise\Cbuilder4\include**algorithm.cc** 完整列表

```
#ifndef __ALGORITH_CC
#define __ALGORITH_CC
#pragma option push -b -a8 -pc -Vx- -Ve- -w-inl -w-aus -w-sig
/*
 *
 * algorithm.cc - Non-inline definitions for the Standard Library algorithms
 *
 *
 * Copyright (c) 1994
 * Hewlett-Packard Company
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Hewlett-Packard Company makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 *
 *
 * (c) Copyright 1994, 1998 Rogue Wave Software, Inc.
 * ALL RIGHTS RESERVED
 *
 * The software and information contained herein are proprietary to, and
 * comprise valuable trade secrets of, Rogue Wave Software, Inc., which
 * intends to preserve as trade secrets such software and information.
 * This software is furnished pursuant to a written license agreement and
 * may be used, copied, transmitted, and stored only in accordance with
 * the terms of such license and with the inclusion of the above copyright
 * notice.  This software and information or any other copies thereof may
 * not be provided or otherwise made available to any other person.
 *
 * Notwithstanding any other lease or license that may pertain to, or
 * accompany the delivery of, this computer software and information, the
 * rights of the Government regarding its use, reproduction and disclosure
 * are as set forth in Section 52.227-19 of the FARS Computer
 * Software-Restricted Rights clause.
 *
 * Use, duplication, or disclosure by the Government is subject to
 * restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in
 * Technical Data and Computer Software clause at DFARS 252.227-7013.
 * Contractor/Manufacturer is Rogue Wave Software, Inc.,
 * P.O. Box 2328, Corvallis, Oregon 97339.
 *
 * This computer software and information is distributed with "restricted
```

```
 * rights."  Use, duplication or disclosure is subject to restrictions as
 * set forth in NASA FAR SUP 18-52.227-79 (April 1985) "Commercial
 * Computer Software-Restricted Rights (April 1985)."  If the Clause at
 * 18-52.227-74 "Rights in Data General" is specified in the contract,
 * then the "Alternate III" clause applies.
 *
 **********************************************************************/

#include <stdcomp.h>

#ifndef _RWSTD_NO_NAMESPACE
namespace std {
#endif

//
// Forward declare raw_storage_iterator
//
  template <class OutputIterator, class T>
  class raw_storage_iterator;

//
// Non-modifying sequence operations.
//

  template <class InputIterator, class Function>
  Function for_each (InputIterator first, InputIterator last, Function f)
  {
    while (first != last) f(*first++);
    return f;
  }

  template <class InputIterator, class T>
  InputIterator find (InputIterator first, InputIterator last, const T& value)
  {
    while (first != last && *first != value)
      ++first;
    return first;
  }

  template <class InputIterator, class Predicate>
  InputIterator find_if (InputIterator first, InputIterator last, Predicate pred)
  {
    while (first != last && !pred(*first)) ++first;
    return first;
  }
  template <class ForwardIterator1, class ForwardIterator2,
  class Distance>
  ForwardIterator1 __find_end (ForwardIterator1 first1,
                               ForwardIterator1 last1,
```

```
                        ForwardIterator2 first2,
                        ForwardIterator2 last2,
                        Distance*)
{
  Distance d, d2;
  __initialize(d,Distance(0));
  __initialize(d2,Distance(0));
  distance(first2,last2,d);
  if (!d)
    return first1;
  distance(first1,last1,d2);
  ForwardIterator1 save = last1;

  while (d2 >= d)
  {
    if (equal(first2,last2,first1))
      save  = first1;
    __initialize(d2,Distance(0));
    distance(++first1,last1,d2);
  }
  return save;
}
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end (ForwardIterator1 first1,
                       ForwardIterator1 last1,
                       ForwardIterator2 first2,
                       ForwardIterator2 last2)
{
  return __find_end(first1,last1,first2,last2,
                 __distance_type(first1));
}

template <class ForwardIterator1, class ForwardIterator2,
class BinaryPredicate, class Distance>
ForwardIterator1 __find_end (ForwardIterator1 first1,
                        ForwardIterator1 last1,
                        ForwardIterator2 first2,
                        ForwardIterator2 last2,
                        BinaryPredicate pred,
                        Distance*)
{
  Distance d, d2;
  __initialize(d,Distance(0));
  __initialize(d2,Distance(0));
  distance(first2,last2,d);
  if (!d)
    return first1;
  distance(first1,last1,d2);
  ForwardIterator1 save = last1;
```

```
  while (d2 >= d)
  {
    if (equal(first2,last2,first1,pred))
      save  = first1;
    __initialize(d2,Distance(0));
    distance(++first1,last1,d2);
  }
  return save;
}

template <class ForwardIterator1, class ForwardIterator2,
class BinaryPredicate>
ForwardIterator1 find_end (ForwardIterator1 first1,
                       ForwardIterator1 last1,
                       ForwardIterator2 first2,
                       ForwardIterator2 last2,
                       BinaryPredicate pred)
{
  return __find_end(first1,last1,first2,last2,
                 pred,__distance_type(first1));
}
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_first_of (ForwardIterator1 first1, ForwardIterator1 last1,
                           ForwardIterator2 first2, ForwardIterator2 last2)
{
  if (first2 == last2)
    return first1;
  ForwardIterator1 next = first1;
  while (next != last1)
  {
    if (find(first2,last2,*next) != last2)
      return next;
    next++;
  }
  return last1;
}

template <class ForwardIterator1, class ForwardIterator2,
class BinaryPredicate>
ForwardIterator1 find_first_of (ForwardIterator1 first1,ForwardIterator1 last1,
                           ForwardIterator2 first2,ForwardIterator2 last2,
                           BinaryPredicate pred)
{
  if (first2 == last2)
    return first1;
  ForwardIterator1 next = first1;
  while (next != last1)
  {
```

```
      if (find_if(first2,last2,bind2nd(pred,*next)) != last2)
        return next;
      next++;
    }
    return last1;
  }
  template <class ForwardIterator>
  ForwardIterator adjacent_find (ForwardIterator first, ForwardIterator last)
  {
    if (first == last) return last;
    ForwardIterator next = first;
    while (++next != last)
    {
      if (*first == *next) return first;
      first = next;
    }
    return last;
  }

  template <class ForwardIterator, class BinaryPredicate>
  ForwardIterator adjacent_find (ForwardIterator first, ForwardIterator last,
                                 BinaryPredicate binary_pred)
  {
    if (first == last) return last;
    ForwardIterator next = first;
    while (++next != last)
    {
      if (binary_pred(*first, *next)) return first;
      first = next;
    }
    return last;
  }

#ifndef _RWSTD_NO_CLASS_PARTIAL_SPEC
  template <class InputIterator, class T>
  _TYPENAME iterator_traits<InputIterator>::difference_type
  count (InputIterator first, InputIterator last, const T& value)
  {
    typename iterator_traits<InputIterator>::difference_type n = 0; //RW_BUG: fix for
bts-42842
    while (first != last)
      if (*first++ == value) ++n;
    return n;
  }

  template <class InputIterator, class Predicate>
  _TYPENAME iterator_traits<InputIterator>::difference_type
  count_if (InputIterator first, InputIterator last, Predicate pred)
  {
```

```
    typename iterator_traits<InputIterator>::difference_type n = 0; //RW_BUG: fix for
bts-42842
    while (first != last)
      if (pred(*first++)) ++n;
    return n;
  }
#endif /* _RWSTD_NO_CLASS_PARTIAL_SPEC */

#ifndef _RWSTD_NO_OLD_COUNT
  template <class InputIterator, class T, class Size>
  void count (InputIterator first, InputIterator last, const T& value, Size& n)
  {
    while (first != last)
      if (*first++ == value) ++n;
  }

  template <class InputIterator, class Predicate, class Size>
  void count_if (InputIterator first, InputIterator last, Predicate pred,
                 Size& n)
  {
    while (first != last)
      if (pred(*first++)) ++n;
  }
#endif /* _RWSTD_NO_OLD_COUNT */

  template <class InputIterator1, class InputIterator2>
  pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1,
                                                InputIterator1 last1,
                                                InputIterator2 first2)
  {
    while (first1 != last1 && *first1 == *first2)
    {
      ++first1;
      ++first2;
    }
    pair<InputIterator1, InputIterator2> tmp(first1, first2);
    return tmp;
  }

  template <class InputIterator1, class InputIterator2, class BinaryPredicate>
  pair<InputIterator1, InputIterator2> mismatch (InputIterator1 first1,
                                                 InputIterator1 last1,
                                                 InputIterator2 first2,
                                                 BinaryPredicate binary_pred)
  {
    while (first1 != last1 && binary_pred(*first1, *first2))
    {
      ++first1;
      ++first2;
```

```
  }
  pair<InputIterator1, InputIterator2> tmp(first1, first2);
  return tmp;
}

template <class ForwardIterator1, class ForwardIterator2,
class Distance1, class Distance2>
ForwardIterator1 __search (ForwardIterator1 first1, ForwardIterator1 last1,
                           ForwardIterator2 first2, ForwardIterator2 last2,
                           Distance1*, Distance2*)
{
  Distance1 d1;
  __initialize(d1, Distance1(0));
  distance(first1, last1, d1);
  Distance2 d2;
  __initialize(d2, Distance2(0));
  distance(first2, last2, d2);

  if (d1 < d2) return last1;

  ForwardIterator1 current1 = first1;
  ForwardIterator2 current2 = first2;

  while (current2 != last2)
  {
    if (*current1++ != *current2++)
      if (d1-- == d2)
        return last1;
      else
      {
        current1 = ++first1;
        current2 = first2;
      }
  }
  return (current2 == last2) ? first1 : last1;
}

template <class ForwardIterator1, class ForwardIterator2,
class BinaryPredicate, class Distance1, class Distance2>
ForwardIterator1 __search (ForwardIterator1 first1, ForwardIterator1 last1,
                           ForwardIterator2 first2, ForwardIterator2 last2,
                           BinaryPredicate binary_pred, Distance1*, Distance2*)
{
  Distance1 d1;
  __initialize(d1, Distance1(0));
  distance(first1, last1, d1);
  Distance2 d2;
  __initialize(d2, Distance2(0));
  distance(first2, last2, d2);
```

```
  if (d1 < d2) return last1;

  ForwardIterator1 current1 = first1;
  ForwardIterator2 current2 = first2;

  while (current2 != last2)
  {
    if (!binary_pred(*current1++, *current2++))
      if (d1-- == d2)
        return last1;
      else
      {
        current1 = ++first1;
        current2 = first2;
      }
  }
  return (current2 == last2) ? first1 : last1;
}

template <class ForwardIterator, class Distance, class Size, class T>
ForwardIterator __search_n (ForwardIterator first, ForwardIterator last,
                            Distance*, Size count, const T& value)
{
  Distance d;
  __initialize(d, Distance(0));
  distance(first, last, d);

  if (d < count || count <= 0) return last;

  Distance       span    = d - count;
  Size          matches = 0;
  ForwardIterator current = first;

  while (current != last)
  {
    if (*current++ != value)
    {
      if (span < matches + 1)
        return last;
      span   -= matches + 1;
      matches = 0;
      first   = current;
    }
    else
      if (++matches == count)
        return first;
  }
```

```
    return last;
  }

  template <class ForwardIterator, class Distance, class Size, class T,
  class BinaryPredicate>
  ForwardIterator __search_n (ForwardIterator first, ForwardIterator last,
                          Distance*, Size count, const T& value,
                          BinaryPredicate pred)
  {
    Distance d;
    __initialize(d, Distance(0));
    distance(first, last, d);

    if (d < count || count <= 0) return last;

    Distance       span    = d - count;
    Size          matches = 0;
    ForwardIterator current = first;

    while (current != last)
    {
      if (!pred(*current++, value))
      {
        if (span < matches + 1)
          return last;
        span   -= matches + 1;
        matches = 0;
        first   = current;
      }
      else
        if (++matches == count)
          return first;
    }

    return last;
  }

//
// Modifying sequence operations.
//

  template <class InputIterator, class OutputIterator>
  OutputIterator copy (InputIterator first, InputIterator last,
                       OutputIterator result)
  {
    while (first != last) *result++ = *first++;
    return result;
  }
```

```
template <class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward (BidirectionalIterator1 first,
                                      BidirectionalIterator1 last,
                                      BidirectionalIterator2 result)
{
  while (first != last) *--result = *--last;
  return result;
}

template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges (ForwardIterator1 first1, ForwardIterator1 last1,
                              ForwardIterator2 first2)
{
  while (first1 != last1) iter_swap(first1++, first2++);
  return first2;
}

template <class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform (InputIterator first, InputIterator last,
                          OutputIterator result, UnaryOperation op)
{
  while (first != last) *result++ = op(*first++);
  return result;
}

template <class InputIterator1, class InputIterator2, class OutputIterator,
class BinaryOperation>
OutputIterator transform (InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, OutputIterator result,
                          BinaryOperation binary_op)
{
  while (first1 != last1) *result++ = binary_op(*first1++, *first2++);
  return result;
}

template <class ForwardIterator, class T>
void replace (ForwardIterator first, ForwardIterator last, const T& old_value,
              const T& new_value)
{
  while (first != last)
  {
    if (*first == old_value) *first = new_value;
    ++first;
  }
}

template <class ForwardIterator, class Predicate, class T>
void replace_if (ForwardIterator first, ForwardIterator last, Predicate pred,
                 const T& new_value)
```

```
  {
    while (first != last)
    {
      if (pred(*first)) *first = new_value;
      ++first;
    }
  }

  template <class InputIterator, class OutputIterator, class T>
  OutputIterator replace_copy (InputIterator first, InputIterator last,
                               OutputIterator result, const T& old_value,
                               const T& new_value)
  {
    while (first != last)
    {
      *result++ = *first == old_value ? new_value : *first;
      ++first;
    }
    return result;
  }

  template <class Iterator, class OutputIterator, class Predicate, class T>
  OutputIterator replace_copy_if (Iterator first, Iterator last,
                                  OutputIterator result, Predicate pred,
                                  const T& new_value)
  {
    while (first != last)
    {
      if(pred(*first))
        *result++ = new_value;
      else
        *result++ = *first;
      ++first;
    }
    return result;
  }

  template <class ForwardIterator, class T>
#ifdef _RWSTD_FILL_NAME_CLASH
  void std_fill (ForwardIterator first, ForwardIterator last, const T& value)
#else
  void fill (ForwardIterator first, ForwardIterator last, const T& value)
#endif
  {
    while (first != last) *first++ = value;
  }

  template <class OutputIterator, class Size, class T>
  void fill_n (OutputIterator first, Size n, const T& value)
```

```
{
  while (n-- > 0) *first++ = value;
}

template <class ForwardIterator, class Generator>
void generate (ForwardIterator first, ForwardIterator last, Generator gen)
{
  while (first != last) *first++ = gen();
}

template <class OutputIterator, class Size, class Generator>
void generate_n (OutputIterator first, Size n, Generator gen)
{
  while (n-- > 0) *first++ = gen();
}

template <class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy (InputIterator first, InputIterator last,
                        OutputIterator result, const T& value)
{
  while (first != last)
  {
    if (*first != value) *result++ = *first;
    ++first;
  }
  return result;
}

template <class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if (InputIterator first, InputIterator last,
                           OutputIterator result, Predicate pred)
{
  while (first != last)
  {
    if (!pred(*first)) *result++ = *first;
    ++first;
  }
  return result;
}

template <class InputIterator, class ForwardIterator>
ForwardIterator __unique_copy (InputIterator first, InputIterator last,
                           ForwardIterator result, forward_iterator_tag)
{
  *result = *first;
  while (++first != last)
    if (*result != *first) *++result = *first;
  return ++result;
}
```

```
template <class InputIterator, class OutputIterator, class T>
OutputIterator __unique_copy (InputIterator first, InputIterator last,
                              OutputIterator result, T*)
{
  T value = *first;
  *result = value;
  while (++first != last)
  {
    if (value != *first)
    {
      value = *first;
      *++result = value;
    }
  }
  return ++result;
}

template <class InputIterator, class ForwardIterator, class BinaryPredicate>
ForwardIterator __unique_copy (InputIterator first, InputIterator last,
                               ForwardIterator result,
                               BinaryPredicate binary_pred,
                               forward_iterator_tag)
{
  *result = *first;
  while (++first != last)
    if (!binary_pred(*result, *first)) *++result = *first;
  return ++result;
}

template <class InputIterator, class OutputIterator, class BinaryPredicate,
class T>
OutputIterator __unique_copy (InputIterator first, InputIterator last,
                              OutputIterator result,
                              BinaryPredicate binary_pred, T*)
{
  T value = *first;
  *result = value;
  while (++first != last)
  {
    if (!binary_pred(value, *first))
    {
      value = *first;
      *++result = value;
    }
  }
  return ++result;
}
```

```
template <class BidirectionalIterator>
void __reverse (BidirectionalIterator first, BidirectionalIterator last,
                bidirectional_iterator_tag)
{
  while (true)
    if (first == last || first == --last)
      return;
    else
      iter_swap(first++, last);
}

template <class RandomAccessIterator>
void __reverse (RandomAccessIterator first, RandomAccessIterator last,
                random_access_iterator_tag)
{
  while (first < last) iter_swap(first++, --last);
}

template <class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy (BidirectionalIterator first,
                             BidirectionalIterator last,
                             OutputIterator result)
{
  while (first != last) *result++ = *--last;
  return result;
}

template <class ForwardIterator, class Distance>
void __rotate (ForwardIterator first, ForwardIterator middle,
               ForwardIterator last, Distance*, forward_iterator_tag)
{
  for (ForwardIterator i = middle; ;)
  {
    iter_swap(first++, i++);
    if (first == middle)
    {
      if (i == last) return;
      middle = i;
    }
    else if (i == last)
      i = middle;
  }
}

template <class EuclideanRingElement>
EuclideanRingElement __gcd (EuclideanRingElement m, EuclideanRingElement n)
{
  while (n != 0)
  {
```

```
      EuclideanRingElement t = m % n;
      m = n;
      n = t;
    }
    return m;
  }

  template <class RandomAccessIterator, class Distance, class T>
  void __rotate_cycle (RandomAccessIterator first, RandomAccessIterator last,
                       RandomAccessIterator initial, Distance shift, T*)
  {
    T value = *initial;
    RandomAccessIterator ptr1 = initial;
    RandomAccessIterator ptr2 = ptr1 + shift;
    while (ptr2 != initial)
    {
      *ptr1 = *ptr2;
      ptr1 = ptr2;
      if (last - ptr2 > shift)
        ptr2 += shift;
      else
        ptr2 = first + (shift - (last - ptr2));
    }
    *ptr1 = value;
  }

  template <class RandomAccessIterator, class Distance>
  void __rotate (RandomAccessIterator first, RandomAccessIterator middle,
                 RandomAccessIterator last, Distance*,
                 random_access_iterator_tag)
  {
    Distance n = __gcd(last - first, middle - first);
    while (n--)
      __rotate_cycle(first, last, first + n, middle - first,
                     _RWSTD_VALUE_TYPE(first));
  }
#ifndef _RWSTD_NO_NAMESPACE
}
namespace __rwstd {
#endif
  extern unsigned _RWSTDExport long __long_random (unsigned long);
#ifndef _RWSTD_NO_NAMESPACE
}
namespace std {
#endif

  template <class RandomAccessIterator, class Distance>
  void __random_shuffle (RandomAccessIterator first, RandomAccessIterator last,
                         Distance*)
```

```
{
  if (!(first == last))
    for (RandomAccessIterator i = first + 1; i != last; ++i)
      iter_swap(i, first + Distance(__RWSTD::__long_random((i - first) + 1)));
}

template <class RandomAccessIterator, class RandomNumberGenerator>
void random_shuffle (RandomAccessIterator first, RandomAccessIterator last,
                     RandomNumberGenerator& rand)
{
  if (!(first == last))
    for (RandomAccessIterator i = first + 1; i != last; ++i)
      iter_swap(i, first + rand((i - first) + 1));
}

template <class BidirectionalIterator, class Predicate>
BidirectionalIterator partition (BidirectionalIterator first,
                                 BidirectionalIterator last, Predicate pred)
{
  while (true)
  {
    while (true)
    {
      if (first == last)
        return first;
      else if (pred(*first))
        ++first;
      else
        break;
    }
    --last;
    while (true)
    {
      if (first == last)
        return first;
      else if (!pred(*last))
        --last;
      else
        break;
    }
    iter_swap(first, last);
    ++first;
  }
}

template <class BidirectionalIterator, class Predicate, class Distance>
BidirectionalIterator __inplace_stable_partition (BidirectionalIterator first,
                                                  BidirectionalIterator last,
                                                  Predicate pred,
```

```
                                          Distance len)
{
  if (len == 1) return pred(*first) ? last : first;
  BidirectionalIterator middle = first;
  advance(middle, len / 2);
  BidirectionalIterator
    first_cut = __inplace_stable_partition(first, middle, pred, len / 2);
  BidirectionalIterator
    second_cut = __inplace_stable_partition(middle, last, pred, len - len / 2);
  rotate(first_cut, middle, second_cut);
  __initialize(len, Distance(0));
  distance(middle, second_cut, len);
  advance(first_cut, len);
  return first_cut;
}

template <class BidirectionalIterator, class Pointer, class Predicate,
class Distance, class T>
BidirectionalIterator __stable_partition_adaptive (BidirectionalIterator first,
                                          BidirectionalIterator last,
                                          Predicate pred, Distance len,
                                          Pointer buffer,
                                          Distance buffer_size,
                                          Distance& fill_pointer, T*)
{
  if (len <= buffer_size)
  {
    len = 0;
    BidirectionalIterator result1 = first;
    Pointer result2 = buffer;
    while (first != last && len < fill_pointer)
    {
      if (pred(*first))
        *result1++ = *first++;
      else
      {
        *result2++ = *first++;
        ++len;
      }
    }
    if (first != last)
    {
      raw_storage_iterator<Pointer, T> result3(result2);
      while (first != last)
      {
        if (pred(*first))
          *result1++ = *first++;
        else
        {
```

```
          *result3++ = *first++;
          ++len;
        }
      }
      fill_pointer = len;
    }
    copy(buffer, buffer + len, result1);
    return result1;
  }
  BidirectionalIterator middle = first;
  advance(middle, len / 2);
  BidirectionalIterator first_cut = __stable_partition_adaptive
    (first, middle, pred, len / 2, buffer, buffer_size, fill_pointer, (T*)0);
  BidirectionalIterator second_cut = __stable_partition_adaptive
    (middle, last, pred, len-len/2, buffer, buffer_size, fill_pointer, (T*)0);
  rotate(first_cut, middle, second_cut);
  __initialize(len, Distance(0));
  distance(middle, second_cut, len);
  advance(first_cut, len);
  return first_cut;
}

template <class BidirectionalIterator, class Predicate, class Pointer,
        class Distance>
BidirectionalIterator __stable_partition (BidirectionalIterator first,
                                  BidirectionalIterator last,
                                  Predicate pred, Distance len,
                                  pair<Pointer, Distance> p)
{
  if (p.first == 0)
    return __inplace_stable_partition(first, last, pred, len);
  Distance fill_pointer = 0;
  BidirectionalIterator result =
    __stable_partition_adaptive(first, last, pred, len, p.first,
                            p.second, fill_pointer,
                            _RWSTD_VALUE_TYPE(first));
  __RWSTD::__destroy(p.first, p.first + fill_pointer);
  return_temporary_buffer(p.first);
  return result;
}

//
// Sorting and related operations.
//

template <class RandomAccessIterator, class T>
RandomAccessIterator __unguarded_partition (RandomAccessIterator first,
                                    RandomAccessIterator last,
                                    T pivot)
```

```
{
  while (true)
  {
    while (*first < pivot) ++first;
    --last;
    while (pivot < *last) --last;
    if (!(first < last)) return first;
    iter_swap(first, last);
    ++first;
  }
}

template <class RandomAccessIterator, class T, class Compare>
RandomAccessIterator __unguarded_partition (RandomAccessIterator first,
                                            RandomAccessIterator last,
                                            T pivot, Compare _RWSTD_COMP)
{
  while (true)
  {
    while (_RWSTD_COMP(*first, pivot)) ++first;
    --last;
    while (_RWSTD_COMP(pivot, *last)) --last;
    if (!(first < last)) return first;
    iter_swap(first, last);
    ++first;
  }
}

const int __stl_threshold = 16;

template <class RandomAccessIterator, class T>
void __quick_sort_loop_aux (RandomAccessIterator first,
                            RandomAccessIterator last, T*)
{
  while (last - first > __stl_threshold)
  {
    RandomAccessIterator cut = __unguarded_partition
    (first, last, T(__median(*first, *(first + (last - first)/2),
                             *(last - 1))));
    if (cut - first >= last - cut)
    {
      __quick_sort_loop(cut, last);
      last = cut;
    }
    else
    {
      __quick_sort_loop(first, cut);
      first = cut;
    }
```

```
    }
  }

  template <class RandomAccessIterator, class T, class Compare>
  void __quick_sort_loop_aux (RandomAccessIterator first,
                          RandomAccessIterator last, T*, Compare _RWSTD_COMP)
  {
    while (last - first > __stl_threshold)
    {
      RandomAccessIterator cut = __unguarded_partition
      (first, last, T(__median(*first, *(first + (last - first)/2),
                           *(last - 1), _RWSTD_COMP)), _RWSTD_COMP);
      if (cut - first >= last - cut)
      {
        __quick_sort_loop(cut, last, _RWSTD_COMP);
        last = cut;
      }
      else
      {
        __quick_sort_loop(first, cut, _RWSTD_COMP);
        first = cut;
      }
    }
  }

  template <class RandomAccessIterator, class T>
  void __unguarded_linear_insert (RandomAccessIterator last, T value)
  {
    RandomAccessIterator next = last;
    --next;
    while (value < *next)
    {
      *last = *next;
      last = next--;
    }
    *last = value;
  }

  template <class RandomAccessIterator, class T, class Compare>
  void __unguarded_linear_insert (RandomAccessIterator last,T value,Compare
_RWSTD_COMP)
  {
    RandomAccessIterator next = last;
    --next;
    while (_RWSTD_COMP(value , *next))
    {
      *last = *next;
      last = next--;
    }
```

```
  *last = value;
}

template <class RandomAccessIterator>
void __insertion_sort (RandomAccessIterator first, RandomAccessIterator last)
{
  if (!(first == last))
    for (RandomAccessIterator i = first + 1; i != last; ++i)
      __linear_insert(first, i, _RWSTD_VALUE_TYPE(first));
}

template <class RandomAccessIterator, class Compare>
void __insertion_sort (RandomAccessIterator first,
                       RandomAccessIterator last, Compare _RWSTD_COMP)
{
  if (!(first == last))
    for (RandomAccessIterator i = first + 1; i != last; ++i)
      __linear_insert(first, i, _RWSTD_VALUE_TYPE(first), _RWSTD_COMP);
}

template <class RandomAccessIterator, class T>
void __unguarded_insertion_sort_aux (RandomAccessIterator first,
                                     RandomAccessIterator last, T*)
{
  for (RandomAccessIterator i = first; i != last; ++i)
    __unguarded_linear_insert(i, T(*i));
}

template <class RandomAccessIterator, class T, class Compare>
void __unguarded_insertion_sort_aux (RandomAccessIterator first,
                                     RandomAccessIterator last,
                                     T*, Compare _RWSTD_COMP)
{
  for (RandomAccessIterator i = first; i != last; ++i)
    __unguarded_linear_insert(i, T(*i), _RWSTD_COMP);
}
template <class RandomAccessIterator>
void __final_insertion_sort (RandomAccessIterator first,
                             RandomAccessIterator last)
{
  if (last - first > __stl_threshold)
  {
    __insertion_sort(first, first + __stl_threshold);
    __unguarded_insertion_sort(first + __stl_threshold, last);
  }
  else
    __insertion_sort(first, last);
}
```

```
template <class RandomAccessIterator, class Compare>
void __final_insertion_sort (RandomAccessIterator first,
                        RandomAccessIterator last, Compare _RWSTD_COMP)
{
  if (last - first > __stl_threshold)
  {
    __insertion_sort(first, first + __stl_threshold, _RWSTD_COMP);
    __unguarded_insertion_sort(first + __stl_threshold, last, _RWSTD_COMP);
  }
  else
    __insertion_sort(first, last, _RWSTD_COMP);
}

template <class RandomAccessIterator1, class RandomAccessIterator2,
class Distance>
void __merge_sort_loop (RandomAccessIterator1 first,
                    RandomAccessIterator1 last,
                    RandomAccessIterator2 result, Distance step_size)
{
  Distance two_step = 2 * step_size;

  while (last - first >= two_step)
  {
    result = merge(first, first + step_size,
                first + step_size, first + two_step, result);
    first += two_step;
  }
  step_size = min(Distance(last - first), step_size);

  merge(first, first + step_size, first + step_size, last, result);
}

template <class RandomAccessIterator1, class RandomAccessIterator2,
class Distance, class Compare>
void __merge_sort_loop (RandomAccessIterator1 first,
                    RandomAccessIterator1 last,
                    RandomAccessIterator2 result, Distance step_size,
                    Compare _RWSTD_COMP)
{
  Distance two_step = 2 * step_size;

  while (last - first >= two_step)
  {
    result = merge(first, first + step_size,
                first + step_size, first + two_step, result, _RWSTD_COMP);
    first += two_step;
  }
  step_size = min(Distance(last - first), step_size);
```

```
  merge(first, first + step_size, first + step_size, last, result, _RWSTD_COMP);
}

const int __stl_chunk_size = 7;

template <class RandomAccessIterator, class Distance>
void __chunk_insertion_sort (RandomAccessIterator first,
                         RandomAccessIterator last, Distance chunk_size)
{
  while (last - first >= chunk_size)
  {
    __insertion_sort(first, first + chunk_size);
    first += chunk_size;
  }
  __insertion_sort(first, last);
}

template <class RandomAccessIterator, class Distance, class Compare>
void __chunk_insertion_sort (RandomAccessIterator first,
                         RandomAccessIterator last,
                         Distance chunk_size, Compare _RWSTD_COMP)
{
  while (last - first >= chunk_size)
  {
    __insertion_sort(first, first + chunk_size, _RWSTD_COMP);
    first += chunk_size;
  }
  __insertion_sort(first, last, _RWSTD_COMP);
}

template <class RandomAccessIterator, class Pointer, class Distance, class T>
void __merge_sort_with_buffer (RandomAccessIterator first,
                           RandomAccessIterator last,
                           Pointer buffer, Distance*, T*)
{
  Distance len = last - first;
  Pointer buffer_last = buffer + len;

  Distance step_size = __stl_chunk_size;
  __chunk_insertion_sort(first, last, step_size);

  while (step_size < len)
  {
    __merge_sort_loop(first, last, buffer, step_size);
    step_size *= 2;
    __merge_sort_loop(buffer, buffer_last, first, step_size);
    step_size *= 2;
  }
}
```

```
template <class RandomAccessIterator, class Pointer, class Distance, class T,
class Compare>
void __merge_sort_with_buffer (RandomAccessIterator first,
                          RandomAccessIterator last, Pointer buffer,
                          Distance*, T*, Compare _RWSTD_COMP)
{
  Distance len = last - first;
  Pointer buffer_last = buffer + len;

  Distance step_size = __stl_chunk_size;
  __chunk_insertion_sort(first, last, step_size, _RWSTD_COMP);

  while (step_size < len)
  {
    __merge_sort_loop(first, last, buffer, step_size, _RWSTD_COMP);
    step_size *= 2;
    __merge_sort_loop(buffer, buffer_last, first, step_size, _RWSTD_COMP);
    step_size *= 2;
  }
}

template <class RandomAccessIterator, class Pointer, class Distance, class T>
void __stable_sort_adaptive (RandomAccessIterator first,
                        RandomAccessIterator last, Pointer buffer,
                        Distance buffer_size, T*)
{
  Distance len = (last - first + 1) / 2;
  RandomAccessIterator middle = first + len;
  if (len > buffer_size)
  {
    __stable_sort_adaptive(first, middle, buffer, buffer_size,
                       _RWSTD_STATIC_CAST(T*,0));
    __stable_sort_adaptive(middle, last, buffer, buffer_size,
                       _RWSTD_STATIC_CAST(T*,0));
  }
  else
  {
    __merge_sort_with_buffer(first, middle, buffer,
                         _RWSTD_STATIC_CAST(Distance*,0),
                         _RWSTD_STATIC_CAST(T*,0));
    __merge_sort_with_buffer(middle, last, buffer,
                         _RWSTD_STATIC_CAST(Distance*,0),
                         _RWSTD_STATIC_CAST(T*,0));
  }
  __merge_adaptive(first, middle, last, Distance(middle - first),
                Distance(last - middle), buffer, buffer_size,
_RWSTD_STATIC_CAST(T*,0));
  }
```

```
  template <class RandomAccessIterator, class Pointer, class Distance, class T,
  class Compare>
  void __stable_sort_adaptive (RandomAccessIterator first,
                          RandomAccessIterator last, Pointer buffer,
                          Distance buffer_size, T*, Compare _RWSTD_COMP)
  {
    Distance len = (last - first + 1) / 2;
    RandomAccessIterator middle = first + len;
    if (len > buffer_size)
    {
      __stable_sort_adaptive(first, middle, buffer,
buffer_size,_RWSTD_STATIC_CAST(T*,0),_RWSTD_COMP);
      __stable_sort_adaptive(middle, last, buffer, buffer_size,
_RWSTD_STATIC_CAST(T*,0),_RWSTD_COMP);
    }
    else
    {
      __merge_sort_with_buffer(first, middle, buffer,
                          _RWSTD_STATIC_CAST(Distance*,0),
                          _RWSTD_STATIC_CAST(T*,0), _RWSTD_COMP);
      __merge_sort_with_buffer(middle, last, buffer,
                          _RWSTD_STATIC_CAST(Distance*,0),
                          _RWSTD_STATIC_CAST(T*,0), _RWSTD_COMP);
    }
    __merge_adaptive(first, middle, last, Distance(middle - first),
                  Distance(last-middle), buffer, buffer_size,
_RWSTD_STATIC_CAST(T*,0),_RWSTD_COMP);
  }

  template <class RandomAccessIterator, class T>
  void __partial_sort (RandomAccessIterator first, RandomAccessIterator middle,
                   RandomAccessIterator last, T*)
  {
    make_heap(first, middle);
    for (RandomAccessIterator i = middle; i < last; ++i)
      if (*i < *first)
        __pop_heap(first, middle, i, T(*i), __distance_type(first));
    sort_heap(first, middle);
  }

  template <class RandomAccessIterator, class T, class Compare>
  void __partial_sort (RandomAccessIterator first, RandomAccessIterator middle,
                   RandomAccessIterator last, T*, Compare _RWSTD_COMP)
  {
    make_heap(first, middle, _RWSTD_COMP);
    for (RandomAccessIterator i = middle; i < last; ++i)
      if (_RWSTD_COMP(*i,*first))
        __pop_heap(first, middle, i, T(*i), _RWSTD_COMP, __distance_type(first));
```

```
  sort_heap(first, middle, _RWSTD_COMP);
}

template <class InputIterator, class RandomAccessIterator, class Distance,
class T>
RandomAccessIterator __partial_sort_copy (InputIterator first,
                                  InputIterator last,
                                  RandomAccessIterator result_first,
                                  RandomAccessIterator result_last,
                                  Distance*, T*)
{
  if (result_first == result_last) return result_last;
  RandomAccessIterator result_real_last = result_first;
  while(first != last && result_real_last != result_last)
    *result_real_last++ = *first++;
  make_heap(result_first, result_real_last);
  while (first != last)
  {
    if (*first < *result_first)
      __adjust_heap(result_first, Distance(0),
                  Distance(result_real_last - result_first),
                  T(*first));
    ++first;
  }
  sort_heap(result_first, result_real_last);
  return result_real_last;
}

template <class InputIterator, class RandomAccessIterator, class Compare,
class Distance, class T>
RandomAccessIterator __partial_sort_copy (InputIterator first,
                                  InputIterator last,
                                  RandomAccessIterator result_first,
                                  RandomAccessIterator result_last,
                                  Compare _RWSTD_COMP, Distance*, T*)
{
  if (result_first == result_last) return result_last;
  RandomAccessIterator result_real_last = result_first;
  while(first != last && result_real_last != result_last)
    *result_real_last++ = *first++;
  make_heap(result_first, result_real_last, _RWSTD_COMP);
  while (first != last)
  {
    if (_RWSTD_COMP(*first,*result_first))
      __adjust_heap(result_first, Distance(0),
                  Distance(result_real_last - result_first), T(*first),
                  _RWSTD_COMP);
    ++first;
  }
```

```
    sort_heap(result_first, result_real_last, _RWSTD_COMP);
    return result_real_last;
  }

  template <class RandomAccessIterator, class T>
  void __nth_element (RandomAccessIterator first, RandomAccessIterator nth,
                      RandomAccessIterator last, T*)
  {
    while (last - first > 3)
    {
      RandomAccessIterator cut = __unguarded_partition
      (first, last, T(__median(*first, *(first + (last - first)/2),
                               *(last - 1))));
      if (cut <= nth)
        first = cut;
      else
        last = cut;
    }
    __insertion_sort(first, last);
  }

  template <class RandomAccessIterator, class T, class Compare>
  void __nth_element (RandomAccessIterator first, RandomAccessIterator nth,
                      RandomAccessIterator last, T*, Compare _RWSTD_COMP)
  {
    while (last - first > 3)
    {
      RandomAccessIterator cut = __unguarded_partition
      (first, last, T(__median(*first, *(first + (last - first)/2),
                               *(last - 1), _RWSTD_COMP)), _RWSTD_COMP);
      if (cut <= nth)
        first = cut;
      else
        last = cut;
    }
    __insertion_sort(first, last, _RWSTD_COMP);
  }

//
// Binary search.
//

  template <class ForwardIterator, class T, class Distance>
  ForwardIterator __lower_bound (ForwardIterator first, ForwardIterator last,
                                 const T& value, Distance*,
                                 forward_iterator_tag)
  {
    Distance len;
    __initialize(len, Distance(0));
```

```
  distance(first, last, len);
  Distance half;
  ForwardIterator middle;

  while (len > 0)
  {
    half = len / 2;
    middle = first;
    advance(middle, half);
    if (*middle < value)
    {
      first = middle;
      ++first;
      len = len - half - 1;
    }
    else
      len = half;
  }
  return first;
}

template <class RandomAccessIterator, class T, class Distance>
RandomAccessIterator __lower_bound (RandomAccessIterator first,
                                    RandomAccessIterator last, const T& value,
                                    Distance*, random_access_iterator_tag)
{
  Distance len = last - first;
  Distance half;
  RandomAccessIterator middle;

  while (len > 0)
  {
    half = len / 2;
    middle = first + half;
    if (*middle < value)
    {
      first = middle + 1;
      len = len - half - 1;
    }
    else
      len = half;
  }
  return first;
}

template <class ForwardIterator, class T, class Compare, class Distance>
ForwardIterator __lower_bound (ForwardIterator first, ForwardIterator last,
                               const T& value, Compare _RWSTD_COMP, Distance*,
                               forward_iterator_tag)
```

```
{
  Distance len;
  __initialize(len, Distance(0));
  distance(first, last, len);
  Distance half;
  ForwardIterator middle;

  while (len > 0)
  {
    half = len / 2;
    middle = first;
    advance(middle, half);
    if (_RWSTD_COMP(*middle, value))
    {
      first = middle;
      ++first;
      len = len - half - 1;
    }
    else
      len = half;
  }
  return first;
}

template <class RandomAccessIterator, class T, class Compare, class Distance>
RandomAccessIterator __lower_bound (RandomAccessIterator first,
                                    RandomAccessIterator last,
                                    const T& value,
                                    Compare _RWSTD_COMP,
                                    Distance*,
                                    random_access_iterator_tag)
{
  Distance len = last - first;
  Distance half;
  RandomAccessIterator middle;

  while (len > 0)
  {
    half = len / 2;
    middle = first + half;
    if (_RWSTD_COMP(*middle, value))
    {
      first = middle + 1;
      len = len - half - 1;
    }
    else
      len = half;
  }
  return first;
```

```
}

template <class ForwardIterator, class T, class Distance>
ForwardIterator __upper_bound (ForwardIterator first, ForwardIterator last,
                          const T& value, Distance*,
                          forward_iterator_tag)
{
  Distance len;
  __initialize(len, Distance(0));
  distance(first, last, len);
  Distance half;
  ForwardIterator middle;

  while (len > 0)
  {
    half = len / 2;
    middle = first;
    advance(middle, half);
    if (value < *middle)
      len = half;
    else
    {
      first = middle;
      ++first;
      len = len - half - 1;
    }
  }
  return first;
}

template <class RandomAccessIterator, class T, class Distance>
RandomAccessIterator __upper_bound (RandomAccessIterator first,
                                RandomAccessIterator last, const T& value,
                                Distance*, random_access_iterator_tag)
{
  Distance len = last - first;
  Distance half;
  RandomAccessIterator middle;

  while (len > 0)
  {
    half = len / 2;
    middle = first + half;
    if (value < *middle)
      len = half;
    else
    {
      first = middle + 1;
      len = len - half - 1;
```

```
    }
  }
  return first;
}

template <class ForwardIterator, class T, class Compare, class Distance>
ForwardIterator __upper_bound (ForwardIterator first, ForwardIterator last,
                           const T& value, Compare _RWSTD_COMP, Distance*,
                           forward_iterator_tag)
{
  Distance len;
  __initialize(len, Distance(0));
  distance(first, last, len);
  Distance half;
  ForwardIterator middle;

  while (len > 0)
  {
    half = len / 2;
    middle = first;
    advance(middle, half);
    if (_RWSTD_COMP(value, *middle))
      len = half;
    else {
      first = middle;
      ++first;
      len = len - half - 1;
    }
  }
  return first;
}

template <class RandomAccessIterator, class T, class Compare, class Distance>
RandomAccessIterator __upper_bound (RandomAccessIterator first,
                                RandomAccessIterator last,
                                const T& value,
                                Compare _RWSTD_COMP, Distance*,
                                random_access_iterator_tag)
{
  Distance len = last - first;
  Distance half;
  RandomAccessIterator middle;

  while (len > 0)
  {
    half = len / 2;
    middle = first + half;
    if (_RWSTD_COMP(value, *middle))
      len = half;
```

```
    else {
      first = middle + 1;
      len = len - half - 1;
    }
  }
  return first;
}

template <class ForwardIterator, class T, class Distance>
pair<ForwardIterator, ForwardIterator>
__equal_range (ForwardIterator first, ForwardIterator last, const T& value,
               Distance*, forward_iterator_tag)
{
  Distance len;
  __initialize(len, Distance(0));
  distance(first, last, len);
  Distance half;
  ForwardIterator middle, left, right;

  while (len > 0)
  {
    half = len / 2;
    middle = first;
    advance(middle, half);
    if (*middle < value)
    {
      first = middle;
      ++first;
      len = len - half - 1;
    }
    else if (value < *middle)
      len = half;
    else
    {
      left = lower_bound(first, middle, value);
      advance(first, len);
      right = upper_bound(++middle, first, value);
      pair<ForwardIterator, ForwardIterator> tmp(left, right);
      return tmp;
    }
  }
  pair<ForwardIterator, ForwardIterator> tmp(first, first);
  return tmp;
}

template <class RandomAccessIterator, class T, class Distance>
pair<RandomAccessIterator, RandomAccessIterator>
__equal_range (RandomAccessIterator first, RandomAccessIterator last,
               const T& value, Distance*, random_access_iterator_tag)
```

```
{
  Distance len = last - first;
  Distance half;
  RandomAccessIterator middle, left, right;

  while (len > 0)
  {
    half = len / 2;
    middle = first + half;
    if (*middle < value)
    {
      first = middle + 1;
      len = len - half - 1;
    }
    else if (value < *middle)
      len = half;
    else
    {
      left = lower_bound(first, middle, value);
      right = upper_bound(++middle, first + len, value);
      pair<RandomAccessIterator,RandomAccessIterator> tmp(left,right);
      return tmp;
    }
  }
  pair<RandomAccessIterator, RandomAccessIterator> tmp(first, first);
  return tmp;
}

template <class ForwardIterator, class T, class Compare, class Distance>
pair<ForwardIterator, ForwardIterator>
__equal_range (ForwardIterator first, ForwardIterator last, const T& value,
             Compare _RWSTD_COMP, Distance*, forward_iterator_tag)
{
  Distance len;
  __initialize(len, Distance(0));
  distance(first, last, len);
  Distance half;
  ForwardIterator middle, left, right;

  while (len > 0)
  {
    half = len / 2;
    middle = first;
    advance(middle, half);
    if (_RWSTD_COMP(*middle, value))
    {
      first = middle;
      ++first;
      len = len - half - 1;
```

```
    }
    else if (_RWSTD_COMP(value, *middle))
      len = half;
    else
    {
      left = lower_bound(first, middle, value, _RWSTD_COMP);
      advance(first, len);
      right = upper_bound(++middle, first, value, _RWSTD_COMP);
      pair<ForwardIterator, ForwardIterator> tmp(left, right);
      return tmp;
    }
  }
  pair<ForwardIterator, ForwardIterator> tmp(first, first);
  return tmp;
}

template <class RandomAccessIterator, class T, class Compare, class Distance>
pair<RandomAccessIterator, RandomAccessIterator>
__equal_range (RandomAccessIterator first, RandomAccessIterator last,
               const T& value, Compare _RWSTD_COMP, Distance*,
               random_access_iterator_tag)
{
  Distance len = last - first;
  Distance half;
  RandomAccessIterator middle, left, right;

  while (len > 0)
  {
    half = len / 2;
    middle = first + half;
    if (_RWSTD_COMP(*middle, value))
    {
      first = middle + 1;
      len = len - half - 1;
    }
    else if (_RWSTD_COMP(value, *middle))
      len = half;
    else
    {
      left = lower_bound(first, middle, value, _RWSTD_COMP);
      right = upper_bound(++middle, first + len, value, _RWSTD_COMP);
      pair<RandomAccessIterator,RandomAccessIterator> tmp(left, right);
      return tmp;
    }
  }
  pair<RandomAccessIterator, RandomAccessIterator> tmp(first, first);
  return tmp;
}
```

```
//
// Merge
//

  template <class InputIterator1, class InputIterator2, class OutputIterator>
  OutputIterator merge (InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result)
  {
    while (first1 != last1 && first2 != last2)
    {
      if (*first2 < *first1)
        *result++ = *first2++;
      else
        *result++ = *first1++;
    }
    return copy(first2, last2, copy(first1, last1, result));
  }

  template <class InputIterator1, class InputIterator2, class OutputIterator,
  class Compare>
  OutputIterator merge (InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result, Compare _RWSTD_COMP)
  {
    while (first1 != last1 && first2 != last2)
    {
      if (_RWSTD_COMP(*first2, *first1))
        *result++ = *first2++;
      else
        *result++ = *first1++;
    }
    return copy(first2, last2, copy(first1, last1, result));
  }

  template <class BidirectionalIterator, class Distance>
  void __merge_without_buffer (BidirectionalIterator first,
                               BidirectionalIterator middle,
                               BidirectionalIterator last,
                               Distance len1, Distance len2)
  {
    if (len1 == 0 || len2 == 0) return;
    if (len1 + len2 == 2)
    {
      if (*middle < *first) iter_swap(first, middle);
      return;
    }
    BidirectionalIterator first_cut = first;
    BidirectionalIterator second_cut = middle;
```

```
  Distance len11;
  __initialize(len11, Distance(0));
  Distance len22;
  __initialize(len22, Distance(0));
  if (len1 > len2)
  {
    len11 = len1 / 2;
    advance(first_cut, len11);
    second_cut = lower_bound(middle, last, *first_cut);
    distance(middle, second_cut, len22);
  }
  else
  {
    len22 = len2 / 2;
    advance(second_cut, len22);
    first_cut = upper_bound(first, middle, *second_cut);
    distance(first, first_cut, len11);
  }
  rotate(first_cut, middle, second_cut);
  BidirectionalIterator new_middle = first_cut;
  advance(new_middle, len22);
  __merge_without_buffer(first, first_cut, new_middle, len11, len22);
  __merge_without_buffer(new_middle, second_cut, last, len1 - len11,
                         len2 - len22);
}

template <class BidirectionalIterator, class Distance, class Compare>
void __merge_without_buffer (BidirectionalIterator first,
                             BidirectionalIterator middle,
                             BidirectionalIterator last,
                             Distance len1, Distance len2, Compare _RWSTD_COMP)
{
  if (len1 == 0 || len2 == 0) return;
  if (len1 + len2 == 2)
  {
    if (_RWSTD_COMP(*middle, *first)) iter_swap(first, middle);
    return;
  }
  BidirectionalIterator first_cut = first;
  BidirectionalIterator second_cut = middle;
  Distance len11;
  __initialize(len11, Distance(0));
  Distance len22;
  __initialize(len22, Distance(0));
  if (len1 > len2)
  {
    len11 = len1 / 2;
    advance(first_cut, len11);
    second_cut = lower_bound(middle, last, *first_cut, _RWSTD_COMP);
```

```
    distance(middle, second_cut, len22);
  }
  else
  {
    len22 = len2 / 2;
    advance(second_cut, len22);
    first_cut = upper_bound(first, middle, *second_cut, _RWSTD_COMP);
    distance(first, first_cut, len11);
  }
  rotate(first_cut, middle, second_cut);
  BidirectionalIterator new_middle = first_cut;
  advance(new_middle, len22);
  __merge_without_buffer(first, first_cut, new_middle, len11, len22, _RWSTD_COMP);
  __merge_without_buffer(new_middle, second_cut, last, len1 - len11,
                         len2 - len22, _RWSTD_COMP);
}

template <class BidirectionalIterator1, class BidirectionalIterator2,
class Distance>
BidirectionalIterator1 __rotate_adaptive (BidirectionalIterator1 first,
                                          BidirectionalIterator1 middle,
                                          BidirectionalIterator1 last,
                                          Distance len1, Distance len2,
                                          BidirectionalIterator2 buffer,
                                          Distance buffer_size)
{
  BidirectionalIterator2 buffer_end;
  if (len1 > len2 && len2 <= buffer_size)
  {
    buffer_end = copy(middle, last, buffer);
    copy_backward(first, middle, last);
    return copy(buffer, buffer_end, first);
  }
  else if (len1 <= buffer_size)
  {
    buffer_end = copy(first, middle, buffer);
    copy(middle, last, first);
    return copy_backward(buffer, buffer_end, last);
  }
  else
  {
    rotate(first, middle, last);
    advance(first, len2);
    return first;
  }
}

template <class BidirectionalIterator1, class BidirectionalIterator2,
class BidirectionalIterator3>
```

```
BidirectionalIterator3 __merge_backward (BidirectionalIterator1 first1,
                                         BidirectionalIterator1 last1,
                                         BidirectionalIterator2 first2,
                                         BidirectionalIterator2 last2,
                                         BidirectionalIterator3 result)
{
  if (first1 == last1) return copy_backward(first2, last2, result);
  if (first2 == last2) return copy_backward(first1, last1, result);
  --last1;
  --last2;
  while (true)
  {
    if (*last2 < *last1)
    {
      *--result = *last1;
      if (first1 == last1) return copy_backward(first2, ++last2, result);
      --last1;
    }
    else
    {
      *--result = *last2;
      if (first2 == last2) return copy_backward(first1, ++last1, result);
      --last2;
    }
  }
}

template <class BidirectionalIterator1, class BidirectionalIterator2,
class BidirectionalIterator3, class Compare>
BidirectionalIterator3 __merge_backward (BidirectionalIterator1 first1,
                                         BidirectionalIterator1 last1,
                                         BidirectionalIterator2 first2,
                                         BidirectionalIterator2 last2,
                                         BidirectionalIterator3 result,
                                         Compare _RWSTD_COMP)
{
  if (first1 == last1) return copy_backward(first2, last2, result);
  if (first2 == last2) return copy_backward(first1, last1, result);
  --last1;
  --last2;
  while (true)
  {
    if (_RWSTD_COMP(*last2, *last1))
    {
      *--result = *last1;
      if (first1 == last1) return copy_backward(first2, ++last2, result);
      --last1;
    }
    else
```

```
      {
        *--result = *last2;
        if (first2 == last2) return copy_backward(first1, ++last1, result);
        --last2;
      }
    }
  }

  template <class BidirectionalIterator, class Distance, class Pointer, class T>
  void __merge_adaptive (BidirectionalIterator first,
                         BidirectionalIterator middle,
                         BidirectionalIterator last, Distance len1,Distance len2,
                         Pointer buffer, Distance buffer_size, T*)
  {
    if (len1 <= len2 && len1 <= buffer_size)
    {
      Pointer end_buffer = copy(first, middle, buffer);
      merge(buffer, end_buffer, middle, last, first);
    }
    else if (len2 <= buffer_size)
    {
      Pointer end_buffer = copy(middle, last, buffer);
      __merge_backward(first, middle, buffer, end_buffer, last);
    }
    else
    {
      BidirectionalIterator first_cut = first;
      BidirectionalIterator second_cut = middle;
      Distance len11;
      __initialize(len11, Distance(0));
      Distance len22;
      __initialize(len22, Distance(0));
      if (len1 > len2)
      {
        len11 = len1 / 2;
        advance(first_cut, len11);
        second_cut = lower_bound(middle, last, *first_cut);
        distance(middle, second_cut, len22);
      }
      else
      {
        len22 = len2 / 2;
        advance(second_cut, len22);
        first_cut = upper_bound(first, middle, *second_cut);
        distance(first, first_cut, len11);
      }
      BidirectionalIterator new_middle =
      __rotate_adaptive(first_cut, middle, second_cut, len1 - len11,
                        len22, buffer, buffer_size);
```

```
    __merge_adaptive(first, first_cut, new_middle, len11, len22, buffer,
                     buffer_size, _RWSTD_STATIC_CAST(T*,0));
    __merge_adaptive(new_middle, second_cut, last, len1 - len11,
                     len2 - len22, buffer, buffer_size, _RWSTD_STATIC_CAST(T*,0));
  }
}

template <class BidirectionalIterator, class Distance, class Pointer, class T,
class Compare>
void __merge_adaptive (BidirectionalIterator first,
                       BidirectionalIterator middle,
                       BidirectionalIterator last, Distance len1,Distance len2,
                       Pointer buffer, Distance buffer_size, T*, Compare _RWSTD_COMP)
{
  if (len1 <= len2 && len1 <= buffer_size)
  {
    Pointer end_buffer = copy(first, middle, buffer);
    merge(buffer, end_buffer, middle, last, first, _RWSTD_COMP);
  }
  else if (len2 <= buffer_size)
  {
    Pointer end_buffer = copy(middle, last, buffer);
    __merge_backward(first, middle, buffer, end_buffer, last, _RWSTD_COMP);
  }
  else
  {
    BidirectionalIterator first_cut = first;
    BidirectionalIterator second_cut = middle;
    Distance len11;
    __initialize(len11, Distance(0));
    Distance len22;
    __initialize(len22, Distance(0));
    if (len1 > len2)
    {
      len11 = len1 / 2;
      advance(first_cut, len11);
      second_cut = lower_bound(middle, last, *first_cut, _RWSTD_COMP);
      distance(middle, second_cut, len22);
    }
    else
    {
      len22 = len2 / 2;
      advance(second_cut, len22);
      first_cut = upper_bound(first, middle, *second_cut, _RWSTD_COMP);
      distance(first, first_cut, len11);
    }
    BidirectionalIterator new_middle =
      __rotate_adaptive(first_cut, middle, second_cut, len1 - len11,
                        len22, buffer, buffer_size);
```

```
      __merge_adaptive(first, first_cut, new_middle, len11, len22, buffer,
                       buffer_size, _RWSTD_STATIC_CAST(T*,0), _RWSTD_COMP);
      __merge_adaptive(new_middle, second_cut, last, len1 - len11,
                       len2 - len22, buffer, buffer_size, _RWSTD_STATIC_CAST(T*,0),
_RWSTD_COMP);
    }
  }

  template <class BidirectionalIterator, class Distance, class Pointer, class T>
  void __inplace_merge (BidirectionalIterator first,
                        BidirectionalIterator middle,
                        BidirectionalIterator last, Distance len1,
                        Distance len2, pair<Pointer, Distance> p, T*)
  {
    if (p.first == 0)
      __merge_without_buffer(first, middle, last, len1, len2);
    else
    {
      Distance len = min(p.second, len1 + len2);
      fill_n(raw_storage_iterator<Pointer, T>(p.first), len, *first);
      __merge_adaptive(first, middle, last, len1, len2, p.first,
                       p.second, _RWSTD_STATIC_CAST(T*,0));
      __RWSTD::__destroy(p.first, p.first + len);
      return_temporary_buffer(p.first);
    }
  }

  template <class BidirectionalIterator, class Distance, class Pointer, class T,
  class Compare>
  void __inplace_merge (BidirectionalIterator first,
                        BidirectionalIterator middle,
                        BidirectionalIterator last, Distance len1,
                        Distance len2, pair<Pointer, Distance> p, T*,
                        Compare _RWSTD_COMP)
  {
    if (p.first == 0)
      __merge_without_buffer(first, middle, last, len1, len2, _RWSTD_COMP);
    else
    {
      Distance len = min(p.second, len1 + len2);
      fill_n(raw_storage_iterator<Pointer, T>(p.first), len, *first);
      __merge_adaptive(first, middle, last, len1, len2, p.first,
                       p.second, _RWSTD_STATIC_CAST(T*,0), _RWSTD_COMP);
      __RWSTD::__destroy(p.first, p.first + len);
      return_temporary_buffer(p.first);
    }
  }

//
```

```
// Set operations.
//

  template <class InputIterator1, class InputIterator2>
  bool includes (InputIterator1 first1, InputIterator1 last1,
                 InputIterator2 first2, InputIterator2 last2)
  {
    while (first1 != last1 && first2 != last2)
    {
      if (*first2 < *first1)
        return false;
      else if(*first1 < *first2)
        ++first1;
      else
        ++first1, ++first2;
    }
    return first2 == last2;
  }

  template <class InputIterator1, class InputIterator2, class Compare>
  bool includes (InputIterator1 first1, InputIterator1 last1,
                 InputIterator2 first2, InputIterator2 last2, Compare _RWSTD_COMP)
  {
    while (first1 != last1 && first2 != last2)
    {
      if (_RWSTD_COMP(*first2, *first1))
        return false;
      else if(_RWSTD_COMP(*first1, *first2))
        ++first1;
      else
        ++first1, ++first2;
    }
    return first2 == last2;
  }

  template <class InputIterator1, class InputIterator2, class OutputIterator>
  OutputIterator set_union (InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2,
                            OutputIterator result)
  {
    while (first1 != last1 && first2 != last2)
    {
      if (*first1 < *first2)
        *result++ = *first1++;
      else if (*first2 < *first1)
        *result++ = *first2++;
      else
      {
        *result++ = *first1++;
```

```
      first2++;
    }
  }
  return copy(first2, last2, copy(first1, last1, result));
}

template <class InputIterator1, class InputIterator2, class OutputIterator,
class Compare>
OutputIterator set_union (InputIterator1 first1, InputIterator1 last1,
                      InputIterator2 first2, InputIterator2 last2,
                      OutputIterator result, Compare _RWSTD_COMP)
{
  while (first1 != last1 && first2 != last2)
  {
    if (_RWSTD_COMP(*first1, *first2))
      *result++ = *first1++;
    else if (_RWSTD_COMP(*first2, *first1))
      *result++ = *first2++;
    else
    {
      *result++ = *first1++;
      ++first2;
    }
  }
  return copy(first2, last2, copy(first1, last1, result));
}

template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_intersection (InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result)
{
  while (first1 != last1 && first2 != last2)
  {
    if (*first1 < *first2)
      ++first1;
    else if (*first2 < *first1)
      ++first2;
    else
    {
      *result++ = *first1++;
      ++first2;
    }
  }
  return result;
}

template <class InputIterator1, class InputIterator2, class OutputIterator,
class Compare>
```

```
OutputIterator set_intersection (InputIterator1 first1, InputIterator1 last1,
                           InputIterator2 first2, InputIterator2 last2,
                           OutputIterator result, Compare _RWSTD_COMP)
{
  while (first1 != last1 && first2 != last2)
  {
    if (_RWSTD_COMP(*first1, *first2))
      ++first1;
    else if (_RWSTD_COMP(*first2, *first1))
      ++first2;
    else
    {
      *result++ = *first1++;
      ++first2;
    }
  }
  return result;
}

template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_difference (InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2,
                          OutputIterator result)
{
  while (first1 != last1 && first2 != last2)
  {
    if (*first1 < *first2)
      *result++ = *first1++;
    else if (*first2 < *first1)
      ++first2;
    else
    {
      ++first1;
      ++first2;
    }
  }
  return copy(first1, last1, result);
}

template <class InputIterator1, class InputIterator2, class OutputIterator,
class Compare>
OutputIterator set_difference (InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2,
                          OutputIterator result, Compare _RWSTD_COMP)
{
  while (first1 != last1 && first2 != last2)
  {
    if (_RWSTD_COMP(*first1, *first2))
      *result++ = *first1++;
```

```
      else if (_RWSTD_COMP(*first2, *first1))
        ++first2;
      else
      {
        ++first1;
        ++first2;
      }
    }
    return copy(first1, last1, result);
  }

  template <class InputIterator1, class InputIterator2, class OutputIterator>
  OutputIterator set_symmetric_difference (InputIterator1 first1,
                                           InputIterator1 last1,
                                           InputIterator2 first2,
                                           InputIterator2 last2,
                                           OutputIterator result)
  {
    while (first1 != last1 && first2 != last2)
    {
      if (*first1 < *first2)
        *result++ = *first1++;
      else if (*first2 < *first1)
        *result++ = *first2++;
      else
      {
        ++first1;
        ++first2;
      }
    }
    return copy(first2, last2, copy(first1, last1, result));
  }

  template <class InputIterator1, class InputIterator2, class OutputIterator,
  class Compare>
  OutputIterator set_symmetric_difference (InputIterator1 first1,
                                           InputIterator1 last1,
                                           InputIterator2 first2,
                                           InputIterator2 last2,
                                           OutputIterator result, Compare _RWSTD_COMP)
  {
    while (first1 != last1 && first2 != last2)
    {
      if (_RWSTD_COMP(*first1, *first2))
        *result++ = *first1++;
      else if (_RWSTD_COMP(*first2, *first1))
        *result++ = *first2++;
      else
      {
```

```
        ++first1;
        ++first2;
      }
    }
    return copy(first2, last2, copy(first1, last1, result));
  }

//
// Heap operations.
//

  template <class RandomAccessIterator, class Distance, class T>
  void __push_heap (RandomAccessIterator first, Distance holeIndex,
                    Distance topIndex, T value)
  {
    Distance parent = (holeIndex - 1) / 2;
    while (holeIndex > topIndex && *(first + parent) < value)
    {
      *(first + holeIndex) = *(first + parent);
      holeIndex = parent;
      parent = (holeIndex - 1) / 2;
    }
    *(first + holeIndex) = value;
  }

  template <class RandomAccessIterator, class Distance, class T, class Compare>
  void __push_heap (RandomAccessIterator first, Distance holeIndex,
                    Distance topIndex, T value, Compare _RWSTD_COMP)
  {
    Distance parent = (holeIndex - 1) / 2;
    while (holeIndex > topIndex && _RWSTD_COMP(*(first + parent), value))
    {
      *(first + holeIndex) = *(first + parent);
      holeIndex = parent;
      parent = (holeIndex - 1) / 2;
    }
    *(first + holeIndex) = value;
  }

  template <class RandomAccessIterator, class Distance, class T>
  void __adjust_heap (RandomAccessIterator first, Distance holeIndex,
                      Distance len, T value)
  {
    Distance topIndex = holeIndex;
    Distance secondChild = 2 * holeIndex + 2;
    while (secondChild < len)
    {
      if (*(first + secondChild) < *(first + (secondChild - 1)))
        secondChild--;
```

```
    *(first + holeIndex) = *(first + secondChild);
    holeIndex = secondChild;
    secondChild = 2 * (secondChild + 1);
  }
  if (secondChild == len)
  {
    *(first + holeIndex) = *(first + (secondChild - 1));
    holeIndex = secondChild - 1;
  }
  __push_heap(first, holeIndex, topIndex, value);
}

template <class RandomAccessIterator, class Distance, class T, class Compare>
void __adjust_heap (RandomAccessIterator first, Distance holeIndex,
                    Distance len, T value, Compare _RWSTD_COMP)
{
  Distance topIndex = holeIndex;
  Distance secondChild = 2 * holeIndex + 2;
  while (secondChild < len)
  {
    if (_RWSTD_COMP(*(first + secondChild), *(first + (secondChild - 1))))
      secondChild--;
    *(first + holeIndex) = *(first + secondChild);
    holeIndex = secondChild;
    secondChild = 2 * (secondChild + 1);
  }
  if (secondChild == len)
  {
    *(first + holeIndex) = *(first + (secondChild - 1));
    holeIndex = secondChild - 1;
  }
  __push_heap(first, holeIndex, topIndex, value, _RWSTD_COMP);
}

template <class RandomAccessIterator, class T, class Distance>
void __make_heap (RandomAccessIterator first, RandomAccessIterator last, T*,
                  Distance*)
{
  Distance len = last - first;
  Distance parent = (len - 2)/2;
  while (true)
  {
    __adjust_heap(first, parent, len, T(*(first + parent)));
    if (parent == 0) return;
    parent--;
  }
}

template <class RandomAccessIterator, class Compare, class T, class Distance>
```

```
  void __make_heap (RandomAccessIterator first, RandomAccessIterator last,
                    Compare _RWSTD_COMP, T*, Distance*)
  {
    Distance len = last - first;
    Distance parent = (len - 2)/2;
    while (true)
    {
      __adjust_heap(first, parent, len, T(*(first + parent)), _RWSTD_COMP);
      if (parent == 0)
        return;
      parent--;
    }
  }

  template <class RandomAccessIterator>
  void sort_heap (RandomAccessIterator first, RandomAccessIterator last)
  {
    while (last - first > 1) pop_heap(first, last--);
  }

  template <class RandomAccessIterator, class Compare>
  void sort_heap (RandomAccessIterator first, RandomAccessIterator last,
                  Compare _RWSTD_COMP)
  {
    while (last - first > 1) pop_heap(first, last--, _RWSTD_COMP);
  }

//
// Minimum and maximum.
//

  template <class ForwardIterator>
  ForwardIterator min_element (ForwardIterator first, ForwardIterator last)
  {
    if (first == last) return first;
    ForwardIterator result = first;
    while (++first != last)
      if (*first < *result) result = first;
    return result;
  }

  template <class ForwardIterator, class Compare>
  ForwardIterator min_element (ForwardIterator first, ForwardIterator last,
                               Compare _RWSTD_COMP)
  {
    if (first == last) return first;
    ForwardIterator result = first;
    while (++first != last)
      if (_RWSTD_COMP(*first, *result)) result = first;
```

```
  return result;
}

template <class ForwardIterator>
ForwardIterator max_element (ForwardIterator first, ForwardIterator last)
{
  if (first == last) return first;
  ForwardIterator result = first;
  while (++first != last)
    if (*result < *first) result = first;
  return result;
}

template <class ForwardIterator, class Compare>
ForwardIterator max_element (ForwardIterator first, ForwardIterator last,
                        Compare _RWSTD_COMP)
{
  if (first == last) return first;
  ForwardIterator result = first;
  while (++first != last)
    if (_RWSTD_COMP(*result, *first)) result = first;
  return result;
}

template <class InputIterator1, class InputIterator2>
bool lexicographical_compare (InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2)
{
  while (first1 != last1 && first2 != last2)
  {
    if (*first1 < *first2)    return true;
    if (*first2++ < *first1++) return false;
  }
  return first1 == last1 && first2 != last2;
}

template <class InputIterator1, class InputIterator2, class Compare>
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        Compare _RWSTD_COMP)
{
  while (first1 != last1 && first2 != last2)
  {
    if (_RWSTD_COMP(*first1, *first2))    return true;
    if (_RWSTD_COMP(*first2++, *first1++)) return false;
  }
  return first1 == last1 && first2 != last2;
}
```

```
//
// Permutations.
//

  template <class BidirectionalIterator>
  bool next_permutation (BidirectionalIterator first,
                         BidirectionalIterator last)
  {
    if (first == last) return false;
    BidirectionalIterator i = first;
    ++i;
    if (i == last) return false;
    i = last;
    --i;

    for (;;)
    {
      BidirectionalIterator ii = i--;
      if (*i < *ii)
      {
        BidirectionalIterator j = last;
        while (!(*i < *--j))
          ;
        iter_swap(i, j);
        reverse(ii, last);
        return true;
      }
      if (i == first)
      {
        reverse(first, last);
        return false;
      }
    }
  }

  template <class BidirectionalIterator, class Compare>
  bool next_permutation (BidirectionalIterator first, BidirectionalIterator last,
                         Compare _RWSTD_COMP)
  {
    if (first == last) return false;
    BidirectionalIterator i = first;
    ++i;
    if (i == last) return false;
    i = last;
    --i;

    for (;;)
    {
      BidirectionalIterator ii = i--;
```

```
      if (_RWSTD_COMP(*i, *ii))
      {
        BidirectionalIterator j = last;
        while (!_RWSTD_COMP(*i, *--j))
          ;
        iter_swap(i, j);
        reverse(ii, last);
        return true;
      }
      if (i == first)
      {
        reverse(first, last);
        return false;
      }
    }
  }

  template <class BidirectionalIterator>
  bool prev_permutation (BidirectionalIterator first,
                         BidirectionalIterator last)
  {
    if (first == last) return false;
    BidirectionalIterator i = first;
    ++i;
    if (i == last) return false;
    i = last;
    --i;

    for (;;)
    {
      BidirectionalIterator ii = i--;
      if (*ii < *i)
      {
        BidirectionalIterator j = last;
        while (!(*--j < *i))
          ;
        iter_swap(i, j);
        reverse(ii, last);
        return true;
      }
      if (i == first)
      {
        reverse(first, last);
        return false;
      }
    }
  }

  template <class BidirectionalIterator, class Compare>
```

```
  bool prev_permutation (BidirectionalIterator first, BidirectionalIterator last,
                         Compare _RWSTD_COMP)
  {
    if (first == last) return false;
    BidirectionalIterator i = first;
    ++i;
    if (i == last) return false;
    i = last;
    --i;

    for(;;)
    {
      BidirectionalIterator ii = i--;
      if (_RWSTD_COMP(*ii, *i))
      {
        BidirectionalIterator j = last;
        while (!_RWSTD_COMP(*--j, *i))
          ;
        iter_swap(i, j);
        reverse(ii, last);
        return true;
      }
      if (i == first)
      {
        reverse(first, last);
        return false;
      }
    }
  }

#ifndef _RWSTD_NO_NAMESPACE
}
#endif

#pragma option pop
#endif /* __ALGORITH_CC */
```