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

```
#ifndef __ALGORITH_H
#define __ALGORITH_H
#pragma option push -b -a8 -pc -Vx- -Ve- -w-inl -w-aus -w-sig
// -*- C++ -*-
#ifndef __STD_ALGORITHM
#define __STD_ALGORITHM

/*
 *
 * algorithm - Declarations and 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_NEW_HEADER
#include <cstdlib>
#else
#include <stdlib.h>
#endif

#include <iterator>
#include <memory>
#include <utility>

// Some compilers have min and max macros
// We use function templates in their stead
#ifdef max
# undef max
# undef __MINMAX_DEFINED  // __BORLANDC__
#endif
#ifdef min
# undef min
# undef __MINMAX_DEFINED  // __BORLANDC__
#endif

#ifndef _RWSTD_NO_NAMESPACE
namespace std {
#endif

//
// Forward declare raw_storage_iterator
//
  template <class OutputIterator, class T>
  class raw_storage_iterator;
  template <class T>
#ifndef __BORLANDC__
  inline
#endif
```

```
  void __initialize (T& t, T val) { t = val; }

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

  template <class InputIterator, class Function>
  Function for_each (InputIterator first, InputIterator last, Function f);

  template <class InputIterator, class T>
  InputIterator find (InputIterator first, InputIterator last, const T& value);

  template <class InputIterator, class Predicate>
  InputIterator find_if (InputIterator first, InputIterator last, Predicate pred);

  template <class ForwardIterator1, class ForwardIterator2,
  class Distance>
  ForwardIterator1 __find_end (ForwardIterator1 first1,
                           ForwardIterator1 last1,
                           ForwardIterator2 first2,
                           ForwardIterator2 last2,
                           Distance*);

  template <class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1 find_end (ForwardIterator1 first1,
                         ForwardIterator1 last1,
                         ForwardIterator2 first2,
                         ForwardIterator2 last2);

  template <class ForwardIterator1, class ForwardIterator2,
  class BinaryPredicate, class Distance>
  ForwardIterator1 __find_end (ForwardIterator1 first1,
                           ForwardIterator1 last1,
                           ForwardIterator2 first2,
                           ForwardIterator2 last2,
                           BinaryPredicate pred,
                           Distance*);

  template <class ForwardIterator1, class ForwardIterator2,
  class BinaryPredicate>
  ForwardIterator1 find_end (ForwardIterator1 first1,
                         ForwardIterator1 last1,
                         ForwardIterator2 first2,
                         ForwardIterator2 last2,
                         BinaryPredicate pred);

  template <class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1 find_first_of (ForwardIterator1 first1, ForwardIterator1 last1,
                              ForwardIterator2 first2, ForwardIterator2 last2);
```

```
  template <class ForwardIterator1, class ForwardIterator2,
  class BinaryPredicate>
  ForwardIterator1 find_first_of (ForwardIterator1 first1,ForwardIterator1 last1,
                            ForwardIterator2 first2,ForwardIterator2 last2,
                            BinaryPredicate pred);

  template <class ForwardIterator>
  ForwardIterator adjacent_find (ForwardIterator first, ForwardIterator last);

  template <class ForwardIterator, class BinaryPredicate>
  ForwardIterator adjacent_find (ForwardIterator first, ForwardIterator last,
                           BinaryPredicate binary_pred);

#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);

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

  template <class InputIterator, class Predicate, class Size>
  void count_if (InputIterator first, InputIterator last, Predicate pred,
              Size& n);
#endif /* _RWSTD_NO_OLD_COUNT */

  template <class InputIterator1, class InputIterator2>
  pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1,
                                        InputIterator1 last1,
                                        InputIterator2 first2);

  template <class InputIterator1, class InputIterator2, class BinaryPredicate>
  pair<InputIterator1, InputIterator2> mismatch (InputIterator1 first1,
                                         InputIterator1 last1,
                                         InputIterator2 first2,
                                         BinaryPredicate binary_pred);

  template <class InputIterator1, class InputIterator2>
  inline bool equal (InputIterator1 first1, InputIterator1 last1,
                 InputIterator2 first2)
  {
    return mismatch(first1, last1, first2).first == last1;
```

```
}

template <class InputIterator1, class InputIterator2, class BinaryPredicate>
inline bool equal (InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, BinaryPredicate binary_pred)
{
  return mismatch(first1, last1, first2, binary_pred).first == last1;
}

template <class ForwardIterator1, class ForwardIterator2,
class Distance1, class Distance2>
ForwardIterator1 __search (ForwardIterator1 first1, ForwardIterator1 last1,
                           ForwardIterator2 first2, ForwardIterator2 last2,
                           Distance1*, Distance2*);

template <class ForwardIterator1, class ForwardIterator2>
inline ForwardIterator1 search (ForwardIterator1 first1,ForwardIterator1 last1,
                                ForwardIterator2 first2,ForwardIterator2 last2)
{
  return __search(first1, last1, first2, last2, __distance_type(first1),
                  __distance_type(first2));
}

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*);

template <class ForwardIterator1, class ForwardIterator2,
class BinaryPredicate>
inline ForwardIterator1 search (ForwardIterator1 first1,ForwardIterator1 last1,
                                ForwardIterator2 first2,ForwardIterator2 last2,
                                BinaryPredicate binary_pred)
{
  return __search(first1, last1, first2, last2, binary_pred,
                  __distance_type(first1), __distance_type(first2));
}

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

template <class ForwardIterator, class Size, class T>
inline ForwardIterator search_n (ForwardIterator first, ForwardIterator last,
                                 Size count, const T& value)
{
  if (count)
    return __search_n(first, last, __distance_type(first), count, value);
```

```
    else
      return first;
  }

  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);

  template <class ForwardIterator, class Size, class T, class BinaryPredicate>
  inline ForwardIterator search_n (ForwardIterator first, ForwardIterator last,
                               Size count, const T& value,
                               BinaryPredicate pred)
  {
    if (count)
      return  __search_n(first, last, __distance_type(first), count,value, pred);
    else
      return first;
  }

//
// Modifying sequence operations.
//

  template <class InputIterator, class OutputIterator>
  OutputIterator copy (InputIterator first, InputIterator last,
                   OutputIterator result);

  template <class BidirectionalIterator1, class BidirectionalIterator2>
  BidirectionalIterator2 copy_backward (BidirectionalIterator1 first,
                                    BidirectionalIterator1 last,
                                    BidirectionalIterator2 result);

  template <class T>
  inline void swap (T& a, T& b)
  {
    T tmp = a;
    a = b;
    b = tmp;
  }

  template <class ForwardIterator1, class ForwardIterator2, class T>
  inline void __iter_swap (ForwardIterator1 a, ForwardIterator2 b, T*)
  {
    T tmp = *a;
    *a = *b;
    *b = tmp;
  }
```

```
  template <class ForwardIterator1, class ForwardIterator2>
  inline void iter_swap (ForwardIterator1 a, ForwardIterator2 b)
  {
    __iter_swap(a, b, _RWSTD_VALUE_TYPE(a));
  }

  template <class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 swap_ranges (ForwardIterator1 first1, ForwardIterator1 last1,
                           ForwardIterator2 first2);

  template <class InputIterator, class OutputIterator, class UnaryOperation>
  OutputIterator transform (InputIterator first, InputIterator last,
                       OutputIterator result, UnaryOperation op);

  template <class InputIterator1, class InputIterator2, class OutputIterator,
  class BinaryOperation>
  OutputIterator transform (InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, OutputIterator result,
                       BinaryOperation binary_op);

  template <class ForwardIterator, class T>
  void replace (ForwardIterator first, ForwardIterator last, const T& old_value,
              const T& new_value);

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

  template <class InputIterator, class OutputIterator, class T>
  OutputIterator replace_copy (InputIterator first, InputIterator last,
                          OutputIterator result, const T& old_value,
                          const T& new_value);

  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);

  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

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

```
template <class ForwardIterator, class Generator>
void generate (ForwardIterator first, ForwardIterator last, Generator gen);

template <class OutputIterator, class Size, class Generator>
void generate_n (OutputIterator first, Size n, Generator gen);

template <class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy (InputIterator first, InputIterator last,
                        OutputIterator result, const T& value);

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

template <class ForwardIterator, class T>
inline ForwardIterator remove (ForwardIterator first, ForwardIterator last,
                           const T& value)
{
  first = find(first, last, value);
  ForwardIterator next = first;
  return first == last ? first : remove_copy(++next, last, first, value);
}

template <class ForwardIterator, class Predicate>
inline ForwardIterator remove_if (ForwardIterator first, ForwardIterator last,
                              Predicate pred)
{
  first = find_if(first, last, pred);
  ForwardIterator next = first;
  return first == last ? first : remove_copy_if(++next, last, first, pred);
}

template <class InputIterator, class ForwardIterator>
ForwardIterator __unique_copy (InputIterator first, InputIterator last,
                           ForwardIterator result, forward_iterator_tag);

template <class InputIterator, class BidirectionalIterator>
inline BidirectionalIterator __unique_copy (InputIterator first,
                                            InputIterator last,
                                            BidirectionalIterator result,
                                            bidirectional_iterator_tag)
{
  return __unique_copy(first, last, result, forward_iterator_tag());
}

template <class InputIterator, class RandomAccessIterator>
inline RandomAccessIterator __unique_copy (InputIterator first,
                                           InputIterator last,
                                           RandomAccessIterator result,
```

```
                                      random_access_iterator_tag)
  {
    return __unique_copy(first, last, result, forward_iterator_tag());
  }

  template <class InputIterator, class OutputIterator, class T>
  OutputIterator __unique_copy (InputIterator first, InputIterator last,
                           OutputIterator result, T*);

  template <class InputIterator, class OutputIterator>
  inline OutputIterator __unique_copy (InputIterator first, InputIterator last,
                                OutputIterator result,
                                output_iterator_tag)
  {
    return __unique_copy(first, last, result, _RWSTD_VALUE_TYPE(first));
  }

  template <class InputIterator, class OutputIterator>
  inline OutputIterator unique_copy (InputIterator first, InputIterator last,
                               OutputIterator result)
  {
    return first == last ? result :
#ifndef _RWSTD_NO_BASE_CLASS_MATCH
    __unique_copy(first, last, result, __iterator_category(result));
#else
    __unique_copy(first, last, result, output_iterator_tag());
#endif
  }

  template <class InputIterator, class ForwardIterator, class BinaryPredicate>
  ForwardIterator __unique_copy (InputIterator first, InputIterator last,
                           ForwardIterator result,
                           BinaryPredicate binary_pred,
                           forward_iterator_tag);
  template <class InputIterator, class BidirectionalIterator,
  class BinaryPredicate>
  inline BidirectionalIterator __unique_copy (InputIterator first,
                                      InputIterator last,
                                      BidirectionalIterator result,
                                      BinaryPredicate binary_pred,
                                      bidirectional_iterator_tag)
  {
    return __unique_copy(first, last, result, binary_pred,
                     forward_iterator_tag());
  }

  template <class InputIterator, class RandomAccessIterator,
  class BinaryPredicate>
  inline RandomAccessIterator __unique_copy (InputIterator first,
```

```
                                    InputIterator last,
                                    RandomAccessIterator result,
                                    BinaryPredicate binary_pred,
                                    random_access_iterator_tag)
  {
    return __unique_copy(first, last, result, binary_pred,
                         forward_iterator_tag());
  }

  template <class InputIterator, class OutputIterator, class BinaryPredicate,
  class T>
  OutputIterator __unique_copy (InputIterator first, InputIterator last,
                                OutputIterator result,
                                BinaryPredicate binary_pred, T*);

  template <class InputIterator, class OutputIterator, class BinaryPredicate>
  inline OutputIterator __unique_copy (InputIterator first, InputIterator last,
                                       OutputIterator result,
                                       BinaryPredicate binary_pred,
                                       output_iterator_tag)
  {
    return __unique_copy(first, last, result, binary_pred,
                         _RWSTD_VALUE_TYPE(first));
  }

  template <class InputIterator, class OutputIterator, class BinaryPredicate>
  inline OutputIterator unique_copy (InputIterator first, InputIterator last,
                                     OutputIterator result,
                                     BinaryPredicate binary_pred)
  {
    return first == last ? result :
#ifndef _RWSTD_NO_BASE_CLASS_MATCH
    __unique_copy(first, last, result, binary_pred, __iterator_category(result));
#else
    __unique_copy(first, last, result, binary_pred, output_iterator_tag());
#endif
  }

  template <class ForwardIterator>
  inline ForwardIterator unique (ForwardIterator first, ForwardIterator last)
  {
    first = adjacent_find(first, last);
    return unique_copy(first, last, first);
  }

  template <class ForwardIterator, class BinaryPredicate>
  inline ForwardIterator unique (ForwardIterator first, ForwardIterator last,
                                 BinaryPredicate binary_pred)
  {
```

```
    first = adjacent_find(first, last, binary_pred);
    return unique_copy(first, last, first, binary_pred);
  }

  template <class BidirectionalIterator>
  void __reverse (BidirectionalIterator first, BidirectionalIterator last,
                  bidirectional_iterator_tag);

  template <class RandomAccessIterator>
  void __reverse (RandomAccessIterator first, RandomAccessIterator last,
                  random_access_iterator_tag);

  template <class BidirectionalIterator>
  inline void reverse (BidirectionalIterator first, BidirectionalIterator last)
  {
#ifndef _RWSTD_NO_BASE_CLASS_MATCH
    __reverse(first, last, __iterator_category(first));
#else
    __reverse(first, last, bidirectional_iterator_tag());
#endif
  }

  template <class BidirectionalIterator, class OutputIterator>
  OutputIterator reverse_copy (BidirectionalIterator first,
                               BidirectionalIterator last,
                               OutputIterator result);

  template <class ForwardIterator, class Distance>
  void __rotate (ForwardIterator first, ForwardIterator middle,
                 ForwardIterator last, Distance*, forward_iterator_tag);

  template <class BidirectionalIterator, class Distance>
  inline void __rotate (BidirectionalIterator first,
                        BidirectionalIterator middle,
                        BidirectionalIterator last, Distance*,
                        bidirectional_iterator_tag)
  {
    reverse(first, middle);
    reverse(middle, last);
    reverse(first, last);
  }

  template <class EuclideanRingElement>
  EuclideanRingElement __gcd (EuclideanRingElement m, EuclideanRingElement n);

  template <class RandomAccessIterator, class Distance, class T>
  void __rotate_cycle (RandomAccessIterator first, RandomAccessIterator last,
                       RandomAccessIterator initial, Distance shift, T*);
```

```
  template <class RandomAccessIterator, class Distance>
  void __rotate (RandomAccessIterator first, RandomAccessIterator middle,
              RandomAccessIterator last, Distance*,
              random_access_iterator_tag);

  template <class ForwardIterator>
  inline void rotate (ForwardIterator first, ForwardIterator middle,
                  ForwardIterator last)
  {
    if (!(first == middle || middle == last))
    {

#ifndef _RWSTD_NO_BASE_CLASS_MATCH
     __rotate(first, middle, last, __distance_type(first),
__iterator_category(first));
#else
     __rotate(first, middle, last, __distance_type(first),
forward_iterator_tag());
#endif
    }
  }

  template <class ForwardIterator, class OutputIterator>
  inline OutputIterator rotate_copy (ForwardIterator first,
                               ForwardIterator middle,
                               ForwardIterator last,
                               OutputIterator result)
  {
    return copy(first, middle, copy(middle, last, result));
  }

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

  template <class RandomAccessIterator>
  inline void random_shuffle (RandomAccessIterator first,
                         RandomAccessIterator last)
  {
    __random_shuffle(first, last, __distance_type(first));
  }

  template <class RandomAccessIterator, class RandomNumberGenerator>
  void random_shuffle (RandomAccessIterator first, RandomAccessIterator last,
                   RandomNumberGenerator& rand);

  template <class BidirectionalIterator, class Predicate>
  BidirectionalIterator partition (BidirectionalIterator first,
                              BidirectionalIterator last, Predicate pred);
```

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

  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*);

  template <class BidirectionalIterator, class Predicate, class Pointer,
  class Distance>
  BidirectionalIterator __stable_partition (BidirectionalIterator first,
                                            BidirectionalIterator last,
                                            Predicate pred, Distance len,
                                            pair<Pointer, Distance> p);

  template <class BidirectionalIterator, class Predicate, class Distance>
  inline BidirectionalIterator __stable_partition_aux (BidirectionalIterator
first,
                                                       BidirectionalIterator last,
                                                       Predicate pred,
                                                       Distance*)
  {
    Distance len;
    __initialize(len, Distance(0));
    distance(first, last, len);

    return len == 0 ? last :
    __stable_partition(first, last, pred, len,
#ifndef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
           get_temporary_buffer<_TYPENAME
              iterator_traits<BidirectionalIterator>::value_type >(len));
#else
             get_temporary_buffer(len,_RWSTD_VALUE_TYPE(first)));
#endif
  }

  template <class BidirectionalIterator, class Predicate>
  inline BidirectionalIterator stable_partition (BidirectionalIterator first,
                                                 BidirectionalIterator last,
                                                 Predicate pred)
  {
```

```
    return __stable_partition_aux(first, last, pred, __distance_type(first));
  }

//
// Sorting and related operations.
//

  template <class T>
  inline const T& __median (const T& a, const T& b, const T& c)
  {
    if (a < b)
      if (b < c)
        return b;
      else if (a < c)
        return c;
      else
        return a;
    else if (a < c)
      return a;
    else if (b < c)
      return c;
    else
      return b;
  }

  template <class T, class Compare>
  inline const T& __median (const T& a, const T& b, const T& c, Compare comp)
  {
    if (comp(a, b))
      if (comp(b, c))
        return b;
      else if (comp(a, c))
        return c;
      else
        return a;
    else if (comp(a, c))
      return a;
    else if (comp(b, c))
      return c;
    else
      return b;
  }

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

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

```
RandomAccessIterator __unguarded_partition (RandomAccessIterator first,
                                            RandomAccessIterator last,
                                            T pivot, Compare comp);

template <class RandomAccessIterator, class T>
void __quick_sort_loop_aux (RandomAccessIterator first,
                            RandomAccessIterator last, T*);

template <class RandomAccessIterator>
inline void __quick_sort_loop (RandomAccessIterator first,
                               RandomAccessIterator last)
{
  __quick_sort_loop_aux(first, last, _RWSTD_VALUE_TYPE(first));
}

template <class RandomAccessIterator, class T, class Compare>
void __quick_sort_loop_aux (RandomAccessIterator first,
                            RandomAccessIterator last, T*, Compare comp);

template <class RandomAccessIterator, class Compare>
inline void __quick_sort_loop (RandomAccessIterator first,
                               RandomAccessIterator last, Compare comp)
{
  __quick_sort_loop_aux(first, last, _RWSTD_VALUE_TYPE(first), comp);
}

template <class RandomAccessIterator, class T>
void __unguarded_linear_insert (RandomAccessIterator last, T value);

template <class RandomAccessIterator, class T, class Compare>
void __unguarded_linear_insert (RandomAccessIterator last,T value,Compare comp);

template <class RandomAccessIterator, class T>
inline void __linear_insert (RandomAccessIterator first,
                             RandomAccessIterator last, T*)
{
  T value = *last;
  if (value < *first)
  {
    copy_backward(first, last, last + 1);
    *first = value;
  }
  else
    __unguarded_linear_insert(last, value);
}

template <class RandomAccessIterator, class T, class Compare>
inline void __linear_insert (RandomAccessIterator first,
                             RandomAccessIterator last, T*, Compare comp)
```

```
{
  T value = *last;
  if (comp(value, *first))
  {
    copy_backward(first, last, last + 1);
    *first = value;
  }
  else
    __unguarded_linear_insert(last, value, comp);
}

template <class RandomAccessIterator>
void __insertion_sort (RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void __insertion_sort (RandomAccessIterator first,
                       RandomAccessIterator last, Compare comp);

template <class RandomAccessIterator, class T>
void __unguarded_insertion_sort_aux (RandomAccessIterator first,
                                     RandomAccessIterator last, T*);

template <class RandomAccessIterator>
inline void __unguarded_insertion_sort(RandomAccessIterator first,
                                       RandomAccessIterator last)
{
  __unguarded_insertion_sort_aux(first, last, _RWSTD_VALUE_TYPE(first));
}

template <class RandomAccessIterator, class T, class Compare>
void __unguarded_insertion_sort_aux (RandomAccessIterator first,
                                     RandomAccessIterator last,
                                     T*, Compare comp);

template <class RandomAccessIterator, class Compare>
inline void __unguarded_insertion_sort (RandomAccessIterator first,
                                        RandomAccessIterator last,
                                        Compare comp)
{
  __unguarded_insertion_sort_aux(first, last, _RWSTD_VALUE_TYPE(first), comp);
}

template <class RandomAccessIterator>
void __final_insertion_sort (RandomAccessIterator first,
                             RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void __final_insertion_sort (RandomAccessIterator first,
                             RandomAccessIterator last, Compare comp);
```

```
template <class RandomAccessIterator>
inline void sort (RandomAccessIterator first, RandomAccessIterator last)
{
  if (!(first == last))
  {
    __quick_sort_loop(first, last);
    __final_insertion_sort(first, last);
  }
}

template <class RandomAccessIterator, class Compare>
inline void sort (RandomAccessIterator first,
                  RandomAccessIterator last, Compare comp)
{
  if (!(first == last))
  {
    __quick_sort_loop(first, last, comp);
    __final_insertion_sort(first, last, comp);
  }
}

template <class RandomAccessIterator>
inline void __inplace_stable_sort (RandomAccessIterator first,
                                   RandomAccessIterator last)
{
  if (last - first < 15)
    __insertion_sort(first, last);
  else
  {
    RandomAccessIterator middle = first + (last - first) / 2;
    __inplace_stable_sort(first, middle);
    __inplace_stable_sort(middle, last);
    __merge_without_buffer(first, middle, last, middle - first,
                           last - middle);
  }
}

template <class RandomAccessIterator, class Compare>
inline void __inplace_stable_sort (RandomAccessIterator first,
                                   RandomAccessIterator last, Compare comp)
{
  if (last - first < 15)
    __insertion_sort(first, last, comp);
  else
  {
    RandomAccessIterator middle = first + (last - first) / 2;
    __inplace_stable_sort(first, middle, comp);
    __inplace_stable_sort(middle, last, comp);
```

```
    __merge_without_buffer(first, middle, last, middle - first,
                           last - middle, comp);
  }
}

template <class RandomAccessIterator1, class RandomAccessIterator2,
class Distance>
void __merge_sort_loop (RandomAccessIterator1 first,
                        RandomAccessIterator1 last,
                        RandomAccessIterator2 result, Distance step_size);

template <class RandomAccessIterator1, class RandomAccessIterator2,
class Distance, class Compare>
void __merge_sort_loop (RandomAccessIterator1 first,
                        RandomAccessIterator1 last,
                        RandomAccessIterator2 result, Distance step_size,
                        Compare comp);

template <class RandomAccessIterator, class Distance>
void __chunk_insertion_sort (RandomAccessIterator first,
                             RandomAccessIterator last, Distance chunk_size);

template <class RandomAccessIterator, class Distance, class Compare>
void __chunk_insertion_sort (RandomAccessIterator first,
                             RandomAccessIterator last,
                             Distance chunk_size, Compare comp);

template <class RandomAccessIterator, class Pointer, class Distance, class T>
void __merge_sort_with_buffer (RandomAccessIterator first,
                               RandomAccessIterator last,
                               Pointer buffer, Distance*, T*);

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 comp);

template <class RandomAccessIterator, class Pointer, class Distance, class T>
void __stable_sort_adaptive (RandomAccessIterator first,
                             RandomAccessIterator last, Pointer buffer,
                             Distance buffer_size, T*);

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 comp);
```

```
  template <class RandomAccessIterator, class Pointer, class Distance, class T>
  inline void __stable_sort (RandomAccessIterator first,
                        RandomAccessIterator last,
                        pair<Pointer, Distance>& p, T*)
  {
    if (p.first == 0)
      __inplace_stable_sort(first, last);
    else
    {
      Distance len = min((int)p.second, (int)(last - first));
      copy(first, first + len, raw_storage_iterator<Pointer, T>(p.first));
      __stable_sort_adaptive(first, last, p.first, p.second,
_RWSTD_STATIC_CAST(T*,0));
      __RWSTD::__destroy(p.first, p.first + len);
      return_temporary_buffer(p.first);
    }
  }

  template <class RandomAccessIterator, class Pointer, class Distance, class T,
  class Compare>
  inline void __stable_sort (RandomAccessIterator first,
                        RandomAccessIterator last,
                        pair<Pointer, Distance>& p, T*, Compare comp)
  {
    if (p.first == 0)
      __inplace_stable_sort(first, last, comp);
    else
    {
      Distance len = min((int)p.second, (int)(last - first));
      copy(first, first + len, raw_storage_iterator<Pointer, T>(p.first));
      __stable_sort_adaptive(first, last, p.first, p.second,
_RWSTD_STATIC_CAST(T*,0), comp);
      __RWSTD::__destroy(p.first, p.first + len);
      return_temporary_buffer(p.first);
    }
  }

  template <class RandomAccessIterator, class T, class Distance>
  inline void __stable_sort_aux (RandomAccessIterator first,
                            RandomAccessIterator last, T*, Distance*)
  {

    pair<T*, Distance> tmp =
#ifndef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
        get_temporary_buffer<T>(Distance(last-first));
#else
        get_temporary_buffer(Distance(last-first),_RWSTD_STATIC_CAST(T*,0));
#endif
    __stable_sort(first, last, tmp, _RWSTD_STATIC_CAST(T*,0));
```

```
  }

  template <class RandomAccessIterator, class T, class Distance, class Compare>
  inline void __stable_sort_aux (RandomAccessIterator first,
                           RandomAccessIterator last, T*, Distance*,
                           Compare comp)
  {
    pair<T*, Distance> tmp =
#ifndef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
       get_temporary_buffer<T>(Distance(last-first));
#else
       get_temporary_buffer(Distance(last-first),_RWSTD_STATIC_CAST(T*,0));
#endif
    __stable_sort(first, last, tmp, _RWSTD_STATIC_CAST(T*,0), comp);
  }

  template <class RandomAccessIterator>
  inline void stable_sort (RandomAccessIterator first,
                       RandomAccessIterator last)
  {
    if (!(first == last))
    {
      __stable_sort_aux(first, last, _RWSTD_VALUE_TYPE(first),
                    __distance_type(first));
    }
  }

  template <class RandomAccessIterator, class Compare>
  inline void stable_sort (RandomAccessIterator first,
                       RandomAccessIterator last, Compare comp)
  {
    if (!(first == last))
    {
      __stable_sort_aux(first, last, _RWSTD_VALUE_TYPE(first),
                    __distance_type(first), comp);
    }
  }

  template <class RandomAccessIterator, class T>
  void __partial_sort (RandomAccessIterator first, RandomAccessIterator middle,
                   RandomAccessIterator last, T*);

  template <class RandomAccessIterator>
  inline void partial_sort (RandomAccessIterator first,
                        RandomAccessIterator middle,
                        RandomAccessIterator last)
  {
    if (!(first == middle))
      __partial_sort(first, middle, last, _RWSTD_VALUE_TYPE(first));
```

```
}

template <class RandomAccessIterator, class T, class Compare>
void __partial_sort (RandomAccessIterator first, RandomAccessIterator middle,
                     RandomAccessIterator last, T*, Compare comp);

template <class RandomAccessIterator, class Compare>
inline void partial_sort (RandomAccessIterator first,
                          RandomAccessIterator middle,
                          RandomAccessIterator last, Compare comp)
{
  if (!(first == middle))
    __partial_sort(first, middle, last, _RWSTD_VALUE_TYPE(first), 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*);

template <class InputIterator, class RandomAccessIterator>
inline RandomAccessIterator
partial_sort_copy (InputIterator first, InputIterator last,
                   RandomAccessIterator result_first,
                   RandomAccessIterator result_last)
{
  return first == last ? result_first :
  __partial_sort_copy(first, last, result_first, result_last,
                      __distance_type(result_first),
                      _RWSTD_VALUE_TYPE(first));
}

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 comp, Distance*, T*);

template <class InputIterator, class RandomAccessIterator, class Compare>
inline RandomAccessIterator
partial_sort_copy (InputIterator first, InputIterator last,
                   RandomAccessIterator result_first,
                   RandomAccessIterator result_last, Compare comp)
{
```

```
  return first == last ? result_first :
  __partial_sort_copy(first, last, result_first, result_last, comp,
                      __distance_type(result_first),
                      _RWSTD_VALUE_TYPE(first));
}

template <class RandomAccessIterator, class T>
void __nth_element (RandomAccessIterator first, RandomAccessIterator nth,
                    RandomAccessIterator last, T*);

template <class RandomAccessIterator>
inline void nth_element (RandomAccessIterator first, RandomAccessIterator nth,
                         RandomAccessIterator last)
{
  if (!(first == last))
    __nth_element(first, nth, last, _RWSTD_VALUE_TYPE(first));
}

template <class RandomAccessIterator, class T, class Compare>
void __nth_element (RandomAccessIterator first, RandomAccessIterator nth,
                    RandomAccessIterator last, T*, Compare comp);

template <class RandomAccessIterator, class Compare>
inline void nth_element (RandomAccessIterator first, RandomAccessIterator nth,
                         RandomAccessIterator last, Compare comp)
{
  if (!(first == last))
    __nth_element(first, nth, last, _RWSTD_VALUE_TYPE(first), comp);
}

//
// Binary search.
//

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

  template <class ForwardIterator, class T, class Distance>
  inline ForwardIterator __lower_bound (ForwardIterator first,
                                        ForwardIterator last,
                                        const T& value, Distance*,
                                        bidirectional_iterator_tag)
  {
    return __lower_bound(first, last, value, _RWSTD_STATIC_CAST(Distance*,0),
                         forward_iterator_tag());
  }
```

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

  template <class ForwardIterator, class T>
  inline ForwardIterator lower_bound (ForwardIterator first,ForwardIterator last,
                                      const T& value)
  {

#ifndef _RWSTD_NO_BASE_CLASS_MATCH
    return __lower_bound(first, last, value, __distance_type(first),
__iterator_category(first));
#else
    return __lower_bound(first, last, value, __distance_type(first),
                         forward_iterator_tag());
#endif
  }

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

  template <class ForwardIterator, class T, class Compare, class Distance>
  inline ForwardIterator __lower_bound (ForwardIterator first,
                                        ForwardIterator last,
                                        const T& value, Compare comp, Distance*,
                                        bidirectional_iterator_tag)
  {
    return __lower_bound(first, last, value, comp,_RWSTD_STATIC_CAST(Distance*,0),
                         forward_iterator_tag());
  }

  template <class RandomAccessIterator, class T, class Compare, class Distance>
  RandomAccessIterator __lower_bound (RandomAccessIterator first,
                                      RandomAccessIterator last,
                                      const T& value, Compare comp, Distance*,
                                      random_access_iterator_tag);

  template <class ForwardIterator, class T, class Compare>
  inline ForwardIterator lower_bound (ForwardIterator first,ForwardIterator last,
                                      const T& value, Compare comp)
  {
#ifndef _RWSTD_NO_BASE_CLASS_MATCH
    return __lower_bound(first, last, value, comp, __distance_type(first),
                         __iterator_category(first));
#else
    return __lower_bound(first, last, value, comp, __distance_type(first),
```

```
                    forward_iterator_tag());
#endif
  }

  template <class ForwardIterator, class T, class Distance>
  ForwardIterator __upper_bound (ForwardIterator first, ForwardIterator last,
                           const T& value, Distance*,
                           forward_iterator_tag);

  template <class ForwardIterator, class T, class Distance>
  inline ForwardIterator __upper_bound (ForwardIterator first,
                                 ForwardIterator last,
                                 const T& value, Distance*,
                                 bidirectional_iterator_tag)
  {
    return __upper_bound(first, last, value, _RWSTD_STATIC_CAST(Distance*,0),
                    forward_iterator_tag());
  }

  template <class RandomAccessIterator, class T, class Distance>
  RandomAccessIterator __upper_bound (RandomAccessIterator first,
                                RandomAccessIterator last, const T& value,
                                Distance*, random_access_iterator_tag);

  template <class ForwardIterator, class T>
  inline ForwardIterator upper_bound (ForwardIterator first,ForwardIterator last,
                                const T& value)
  {
#ifndef _RWSTD_NO_BASE_CLASS_MATCH
    return __upper_bound(first, last, value, __distance_type(first),
                    __iterator_category(first));
#else
    return __upper_bound(first, last, value, __distance_type(first),
                    forward_iterator_tag());
#endif
  }

  template <class ForwardIterator, class T, class Compare, class Distance>
  ForwardIterator __upper_bound (ForwardIterator first, ForwardIterator last,
                           const T& value, Compare comp, Distance*,
                           forward_iterator_tag);

  template <class ForwardIterator, class T, class Compare, class Distance>
  inline ForwardIterator __upper_bound (ForwardIterator first,
                                 ForwardIterator last,
                                 const T& value, Compare comp, Distance*,
                                 bidirectional_iterator_tag)
  {
    return __upper_bound(first, last, value, comp, _RWSTD_STATIC_CAST(Distance*,0),
```

```
                    forward_iterator_tag());
  }

  template <class RandomAccessIterator, class T, class Compare, class Distance>
  RandomAccessIterator __upper_bound (RandomAccessIterator first,
                                RandomAccessIterator last,
                                const T& value, Compare comp, Distance*,
                                random_access_iterator_tag);

  template <class ForwardIterator, class T, class Compare>
  inline ForwardIterator upper_bound (ForwardIterator first,ForwardIterator last,
                                const T& value, Compare comp)
  {
#ifndef _RWSTD_NO_BASE_CLASS_MATCH
    return __upper_bound(first, last, value, comp, __distance_type(first),
                    __iterator_category(first));
#else
    return __upper_bound(first, last, value, comp, __distance_type(first),
                    forward_iterator_tag());
#endif
  }

  template <class ForwardIterator, class T, class Distance>
  pair<ForwardIterator, ForwardIterator>
  __equal_range (ForwardIterator first, ForwardIterator last, const T& value,
              Distance*, forward_iterator_tag);

  template <class ForwardIterator, class T, class Distance>
  inline pair<ForwardIterator, ForwardIterator>
  __equal_range (ForwardIterator first, ForwardIterator last, const T& value,
              Distance*, bidirectional_iterator_tag)
  {
    return __equal_range(first, last, value, _RWSTD_STATIC_CAST(Distance*,0),
                    forward_iterator_tag());
  }

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

  template <class ForwardIterator, class T>
  inline pair<ForwardIterator, ForwardIterator>
  equal_range (ForwardIterator first, ForwardIterator last, const T& value)
  {
#ifndef _RWSTD_NO_BASE_CLASS_MATCH
    return __equal_range(first, last, value, __distance_type(first),
                    __iterator_category(first));
#else
```

```
    return __equal_range(first, last, value, __distance_type(first),
                         forward_iterator_tag());
#endif
  }

  template <class ForwardIterator, class T, class Compare, class Distance>
  pair<ForwardIterator, ForwardIterator>
  __equal_range (ForwardIterator first, ForwardIterator last, const T& value,
                 Compare comp, Distance*, forward_iterator_tag);

  template <class ForwardIterator, class T, class Compare, class Distance>
  inline pair<ForwardIterator, ForwardIterator>
  __equal_range (ForwardIterator first, ForwardIterator last, const T& value,
                 Compare comp, Distance*, bidirectional_iterator_tag)
  {
    return __equal_range(first, last, value, comp, _RWSTD_STATIC_CAST(Distance*,0),
                         forward_iterator_tag());
  }

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

  template <class ForwardIterator, class T, class Compare>
  inline pair<ForwardIterator, ForwardIterator>
  equal_range (ForwardIterator first, ForwardIterator last, const T& value,
               Compare comp)
  {
#ifndef _RWSTD_NO_BASE_CLASS_MATCH
    return __equal_range(first, last, value, comp, __distance_type(first),
                         __iterator_category(first));
#else
    return __equal_range(first, last, value, comp, __distance_type(first),
                         forward_iterator_tag());
#endif
  }

  template <class ForwardIterator, class T>
  inline bool binary_search (ForwardIterator first, ForwardIterator last,
                             const T& value)
  {
    ForwardIterator i = lower_bound(first, last, value);
    return i != last && !(value < *i);
  }

  template <class ForwardIterator, class T, class Compare>
  inline bool binary_search (ForwardIterator first, ForwardIterator last,
```

```
                        const T& value, Compare comp)
  {
    ForwardIterator i = lower_bound(first, last, value, comp);
    return i != last && !comp(value, *i);
  }

//
// Merge
//

  template <class InputIterator1, class InputIterator2, class OutputIterator>
  OutputIterator merge (InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, InputIterator2 last2,
                    OutputIterator result);

  template <class InputIterator1, class InputIterator2, class OutputIterator,
  class Compare>
  OutputIterator merge (InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, InputIterator2 last2,
                    OutputIterator result, Compare comp);

  template <class BidirectionalIterator, class Distance>
  void __merge_without_buffer (BidirectionalIterator first,
                          BidirectionalIterator middle,
                          BidirectionalIterator last,
                          Distance len1, Distance len2);

  template <class BidirectionalIterator, class Distance, class Compare>
  void __merge_without_buffer (BidirectionalIterator first,
                          BidirectionalIterator middle,
                          BidirectionalIterator last,
                          Distance len1, Distance len2, Compare 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);

  template <class BidirectionalIterator1, class BidirectionalIterator2,
  class BidirectionalIterator3>
  BidirectionalIterator3 __merge_backward (BidirectionalIterator1 first1,
                                     BidirectionalIterator1 last1,
                                     BidirectionalIterator2 first2,
                                     BidirectionalIterator2 last2,
                                     BidirectionalIterator3 result);
```

```
template <class BidirectionalIterator1, class BidirectionalIterator2,
class BidirectionalIterator3, class Compare>
BidirectionalIterator3 __merge_backward (BidirectionalIterator1 first1,
                                         BidirectionalIterator1 last1,
                                         BidirectionalIterator2 first2,
                                         BidirectionalIterator2 last2,
                                         BidirectionalIterator3 result,
                                         Compare comp);

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*);

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 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*);

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 comp);

template <class BidirectionalIterator, class T, class Distance>
inline void __inplace_merge_aux (BidirectionalIterator first,
                                 BidirectionalIterator middle,
                                 BidirectionalIterator last, T*, Distance*)
{
  Distance len1;
  __initialize(len1, Distance(0));
  distance(first, middle, len1);
  Distance len2;
  __initialize(len2, Distance(0));
  distance(middle, last, len2);
  __inplace_merge(first, middle, last, len1, len2,
```

```
#ifndef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
       get_temporary_buffer<T>(len1+len2),_RWSTD_STATIC_CAST(T*,0));
#else

get_temporary_buffer(len1+len2,_RWSTD_STATIC_CAST(T*,0)),_RWSTD_STATIC_CAST(T*,
0));
#endif

  }

  template <class BidirectionalIterator, class T, class Distance, class Compare>
  inline void __inplace_merge_aux (BidirectionalIterator first,
                                   BidirectionalIterator middle,
                                   BidirectionalIterator last, T*, Distance*,
                                   Compare comp)
  {
    Distance len1;
    __initialize(len1, Distance(0));
    distance(first, middle, len1);
    Distance len2;
    __initialize(len2, Distance(0));
    distance(middle, last, len2);
    __inplace_merge(first, middle, last, len1, len2,
#ifndef _RWSTD_NO_TEMPLATE_ON_RETURN_TYPE
       get_temporary_buffer<T>(len1+len2), _RWSTD_STATIC_CAST(T*,0), comp);
#else
       get_temporary_buffer(len1 + len2, _RWSTD_STATIC_CAST(T*,0)),
_RWSTD_STATIC_CAST(T*,0), comp);
#endif
  }

  template <class BidirectionalIterator>
  inline void inplace_merge (BidirectionalIterator first,
                             BidirectionalIterator middle,
                             BidirectionalIterator last)
  {
    if (!(first == middle || middle == last))
      __inplace_merge_aux(first, middle, last, _RWSTD_VALUE_TYPE(first),
                          __distance_type(first));
  }

  template <class BidirectionalIterator, class Compare>
  inline void inplace_merge (BidirectionalIterator first,
                             BidirectionalIterator middle,
                             BidirectionalIterator last, Compare comp)
  {
    if (!(first == middle || middle == last))
      __inplace_merge_aux(first, middle, last, _RWSTD_VALUE_TYPE(first),
                          __distance_type(first), comp);
```

```
  }

//
// Set operations.
//

  template <class InputIterator1, class InputIterator2>
  bool includes (InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2);

  template <class InputIterator1, class InputIterator2, class Compare>
  bool includes (InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              Compare comp);

  template <class InputIterator1, class InputIterator2, class OutputIterator>
  OutputIterator set_union (InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result);

  template <class InputIterator1, class InputIterator2, class OutputIterator,
  class Compare>
  OutputIterator set_union (InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result, Compare comp);

  template <class InputIterator1, class InputIterator2, class OutputIterator>
  OutputIterator set_intersection (InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result);

  template <class InputIterator1, class InputIterator2, class OutputIterator,
  class Compare>
  OutputIterator set_intersection (InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result, Compare comp);

  template <class InputIterator1, class InputIterator2, class OutputIterator>
  OutputIterator set_difference (InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2,
                            OutputIterator result);

  template <class InputIterator1, class InputIterator2, class OutputIterator,
  class Compare>
  OutputIterator set_difference (InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2,
                            OutputIterator result, Compare comp);

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

```
  OutputIterator set_symmetric_difference (InputIterator1 first1,
                                           InputIterator1 last1,
                                           InputIterator2 first2,
                                           InputIterator2 last2,
                                           OutputIterator 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 comp);

//
// Heap operations.
//

  template <class RandomAccessIterator, class Distance, class T>
  void __push_heap (RandomAccessIterator first, Distance holeIndex,
                Distance topIndex, T value);

  template <class RandomAccessIterator, class Distance, class T>
  inline void __push_heap_aux (RandomAccessIterator first,
                           RandomAccessIterator last, Distance*, T*)
  {
    __push_heap(first, Distance((last-first)-1), Distance(0), T(*(last-1)));
  }

  template <class RandomAccessIterator>
  inline void push_heap (RandomAccessIterator first, RandomAccessIterator last)
  {
    if (!(first == last))
      __push_heap_aux(first, last, __distance_type(first),
                  _RWSTD_VALUE_TYPE(first));
  }

  template <class RandomAccessIterator, class Distance, class T, class Compare>
  void __push_heap (RandomAccessIterator first, Distance holeIndex,
                Distance topIndex, T value, Compare comp);

  template <class RandomAccessIterator, class Compare,  class Distance, class T>
  inline void __push_heap_aux (RandomAccessIterator first,
                           RandomAccessIterator last, Compare comp,
                           Distance*, T*)
  {
    __push_heap(first, Distance((last-first)-1), Distance(0),
              T(*(last - 1)), comp);
```

```
}

template <class RandomAccessIterator, class Compare>
inline void push_heap (RandomAccessIterator first, RandomAccessIterator last,
                       Compare comp)
{
  if (!(first == last))
    __push_heap_aux(first, last, comp, __distance_type(first),
                    _RWSTD_VALUE_TYPE(first));
}

template <class RandomAccessIterator, class Distance, class T>
void __adjust_heap (RandomAccessIterator first, Distance holeIndex,
                    Distance len, T value);

template <class RandomAccessIterator, class T, class Distance>
inline void __pop_heap (RandomAccessIterator first, RandomAccessIterator last,
                        RandomAccessIterator result, T value, Distance*)
{
  *result = *first;
  __adjust_heap(first, Distance(0), Distance(last - first), value);
}

template <class RandomAccessIterator, class T>
inline void __pop_heap_aux (RandomAccessIterator first,
                            RandomAccessIterator last, T*)
{
  __pop_heap(first, last-1, last-1, T(*(last-1)), __distance_type(first));
}

template <class RandomAccessIterator>
inline void pop_heap (RandomAccessIterator first, RandomAccessIterator last)
{
  if (!(first == last))
    __pop_heap_aux(first, last, _RWSTD_VALUE_TYPE(first));
}

template <class RandomAccessIterator, class Distance, class T, class Compare>
void __adjust_heap (RandomAccessIterator first, Distance holeIndex,
                    Distance len, T value, Compare comp);

template <class RandomAccessIterator, class T, class Compare, class Distance>
inline void __pop_heap (RandomAccessIterator first, RandomAccessIterator last,
                        RandomAccessIterator result, T value, Compare comp,
                        Distance*)
{
  *result = *first;
  __adjust_heap(first, Distance(0), Distance(last - first), value, comp);
}
```

```
template <class RandomAccessIterator, class T, class Compare>
inline void __pop_heap_aux (RandomAccessIterator first,
                        RandomAccessIterator last, T*, Compare comp)
{
  __pop_heap(first, last - 1, last - 1, T(*(last - 1)), comp,
             __distance_type(first));
}

template <class RandomAccessIterator, class Compare>
inline void pop_heap (RandomAccessIterator first, RandomAccessIterator last,
                      Compare comp)
{
  if (!(first == last))
    __pop_heap_aux(first, last, _RWSTD_VALUE_TYPE(first), comp);
}

template <class RandomAccessIterator, class T, class Distance>
void __make_heap (RandomAccessIterator first, RandomAccessIterator last, T*,
                  Distance*);

template <class RandomAccessIterator>
inline void make_heap (RandomAccessIterator first, RandomAccessIterator last)
{
  if (!(last - first < 2))
    __make_heap(first, last, _RWSTD_VALUE_TYPE(first),
                __distance_type(first));
}

template <class RandomAccessIterator, class Compare, class T, class Distance>
void __make_heap (RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp, T*, Distance*);

template <class RandomAccessIterator, class Compare>
inline void make_heap (RandomAccessIterator first, RandomAccessIterator last,
                       Compare comp)
{
  if (!(last - first < 2))
    __make_heap(first, last, comp, _RWSTD_VALUE_TYPE(first),
                __distance_type(first));
}

template <class RandomAccessIterator>
void sort_heap (RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void sort_heap (RandomAccessIterator first, RandomAccessIterator last,
                Compare comp);
```

```
//
// Minimum and maximum.
//

#if !defined(__MINMAX_DEFINED)
  template <class T>
  inline const T& min (const T& a, const T& b)
  {
    return b < a ? b : a;
  }
#endif

  template <class T, class Compare>
  inline const T& min (const T& a, const T& b, Compare comp)
  {
    return comp(b, a) ? b : a;
  }

#if !defined(__MINMAX_DEFINED)
  template <class T>
  inline const T& max (const T& a, const T& b)
  {
    return  a < b ? b : a;
  }
#endif

  template <class T, class Compare>
  inline const T& max (const T& a, const T& b, Compare comp)
  {
    return comp(a, b) ? b : a;
  }

  template <class ForwardIterator>
  ForwardIterator min_element (ForwardIterator first, ForwardIterator last);

  template <class ForwardIterator, class Compare>
  ForwardIterator min_element (ForwardIterator first, ForwardIterator last,
                               Compare comp);

  template <class ForwardIterator>
  ForwardIterator max_element (ForwardIterator first, ForwardIterator last);

  template <class ForwardIterator, class Compare>
  ForwardIterator max_element (ForwardIterator first, ForwardIterator last,
                               Compare comp);

  template <class InputIterator1, class InputIterator2>
  bool lexicographical_compare (InputIterator1 first1, InputIterator1 last1,
                                InputIterator2 first2, InputIterator2 last2);
```

```
  template <class InputIterator1, class InputIterator2, class Compare>
  bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                               InputIterator2 first2, InputIterator2 last2,
                               Compare comp);

//
// Permutations.
//

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

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

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

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

#ifndef _RWSTD_NO_NAMESPACE
}
#endif

#ifdef _RWSTD_NO_TEMPLATE_REPOSITORY
#include <algorith.cc>
#endif

#ifndef __USING_STD_NAMES__
  using namespace std;
#endif
#endif /*__STD_ALGORITHM*/
#pragma option pop
#endif /* __ALGORITH_H */
```