G++ 2.91.57，cygnus\cygwin-b20\include\g++\stl_deque.h 完整列表

```
/*
 *
 * 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.
 *
 *
 * Copyright (c) 1997
 * Silicon Graphics Computer Systems, Inc.
 *
 * 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.  Silicon Graphics makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 */

/* NOTE: This is an internal header file, included by other STL headers.
 *   You should not attempt to use it directly.
 */

#ifndef __SGI_STL_INTERNAL_DEQUE_H
#define __SGI_STL_INTERNAL_DEQUE_H

/* Class invariants:
 *  For any nonsingular iterator i:
 *    i.node is the address of an element in the map array.  The
 *      contents of i.node is a pointer to the beginning of a node.
 *    i.first == *(i.node)
 *    i.last  == i.first + node_size
 *    i.cur is a pointer in the range [i.first, i.last).  NOTE:
 *      the implication of this is that i.cur is always a dereferenceable
 *      pointer, even if i is a past-the-end iterator.
 *  Start and Finish are always nonsingular iterators.  NOTE: this means
 *    that an empty deque must have one node, and that a deque
 *    with N elements, where N is the buffer size, must have two nodes.
 *  For every node other than start.node and finish.node, every element
 *    in the node is an initialized object.  If start.node == finish.node,
 *    then [start.cur, finish.cur) are initialized objects, and
 *    the elements outside that range are uninitialized storage.  Otherwise,
 *    [start.cur, start.last) and [finish.first, finish.cur) are initialized
 *    objects, and [start.first, start.cur) and [finish.cur, finish.last)
 *    are uninitialized storage.
 *  [map, map + map_size) is a valid, non-empty range.
 *  [start.node, finish.node] is a valid range contained within
 *    [map, map + map_size).
 *  A pointer in the range [map, map + map_size) points to an allocated
 *    node if and only if the pointer is in the range [start.node, finish.node].
 */


/*
 * In previous versions of deque, node_size was fixed by the
 * implementation.  In this version, however, users can select
 * the node size.  Deque has three template parameters; the third,
 * a number of type size_t, is the number of elements per node.
 * If the third template parameter is 0 (which is the default),
 * then deque will use a default node size.
 *
 * The only reason for using an alternate node size is if your application
 * requires a different performance tradeoff than the default.  If,
 * for example, your program contains many deques each of which contains
 * only a few elements, then you might want to save memory (possibly
 * by sacrificing some speed) by using smaller nodes.
 *
 * Unfortunately, some compilers have trouble with non-type template
 * parameters; stl_config.h defines __STL_NON_TYPE_TMPL_PARAM_BUG if
 * that is the case.  If your compiler is one of them, then you will
 * not be able to use alternate node sizes; you will have to use the
 * default value.
 */

__STL_BEGIN_NAMESPACE

#if defined(__sgi) && !defined(__GNUC__) && (_MIPS_SIM != _MIPS_SIM_ABI32)
#pragma set woff 1174
#endif

// Note: this function is simply a kludge to work around several compilers'
//  bugs in handling constant expressions.
inline size_t __deque_buf_size(size_t n, size_t sz)
{
  return n != 0 ? n : (sz < 512 ? size_t(512 / sz) : size_t(1));
}

#ifndef __STL_NON_TYPE_TMPL_PARAM_BUG
template <class T, class Ref, class Ptr, size_t BufSiz>
```

```
struct __deque_iterator {
  typedef __deque_iterator<T, T&, T*, BufSiz>           iterator;
  typedef __deque_iterator<T, const T&, const T*, BufSiz> const_iterator;
  static size_t buffer_size() {return __deque_buf_size(BufSiz, sizeof(T)); }
#else /* __STL_NON_TYPE_TMPL_PARAM_BUG */
template <class T, class Ref, class Ptr>
struct __deque_iterator {
  typedef __deque_iterator<T, T&, T*>           iterator;
  typedef __deque_iterator<T, const T&, const T*> const_iterator;
  static size_t buffer_size() {return __deque_buf_size(0, sizeof(T)); }
#endif

  typedef random_access_iterator_tag iterator_category;
  typedef T value_type;
  typedef Ptr pointer;
  typedef Ref reference;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;
  typedef T** map_pointer;

  typedef __deque_iterator self;

  T* cur;
  T* first;
  T* last;
  map_pointer node;

  __deque_iterator(T* x, map_pointer y)
    : cur(x), first(*y), last(*y + buffer_size()), node(y) {}
  __deque_iterator() : cur(0), first(0), last(0), node(0) {}
  __deque_iterator(const iterator& x)
    : cur(x.cur), first(x.first), last(x.last), node(x.node) {}

  reference operator*() const { return *cur; }
#ifndef __SGI_STL_NO_ARROW_OPERATOR
  pointer operator->() const { return &(operator*()); }
#endif /* __SGI_STL_NO_ARROW_OPERATOR */

  difference_type operator-(const self& x) const {
    return difference_type(buffer_size()) * (node - x.node - 1) +
      (cur - first) + (x.last - x.cur);
  }

  self& operator++() {
    ++cur;
    if (cur == last) {
      set_node(node + 1);
      cur = first;
    }
    return *this;
  }
  self operator++(int)  {
    self tmp = *this;
    ++*this;
    return tmp;
  }

  self& operator--() {
    if (cur == first) {
      set_node(node - 1);
      cur = last;
    }
    --cur;
    return *this;
  }
  self operator--(int) {
    self tmp = *this;
    --*this;
    return tmp;
  }

  self& operator+=(difference_type n) {
    difference_type offset = n + (cur - first);
    if (offset >= 0 && offset < difference_type(buffer_size()))
      cur += n;
    else {
      difference_type node_offset =
        offset > 0 ? offset / difference_type(buffer_size())
                   : -difference_type((-offset - 1) / buffer_size()) - 1;
      set_node(node + node_offset);
      cur = first + (offset - node_offset * difference_type(buffer_size()));
    }
    return *this;
  }

  self operator+(difference_type n) const {
    self tmp = *this;
    return tmp += n;
  }

  self& operator-=(difference_type n) { return *this += -n; }

  self operator-(difference_type n) const {
    self tmp = *this;
    return tmp -= n;
  }

  reference operator[](difference_type n) const { return *(*this + n); }
  bool operator==(const self& x) const { return cur == x.cur; }
  bool operator!=(const self& x) const { return !(*this == x); }
  bool operator<(const self& x) const {
    return (node == x.node) ? (cur < x.cur) : (node < x.node);
  }

  void set_node(map_pointer new_node) {
    node = new_node;
    first = *new_node;
    last = first + difference_type(buffer_size());
  }
};

#ifndef __STL_CLASS_PARTIAL_SPECIALIZATION

#ifndef __STL_NON_TYPE_TMPL_PARAM_BUG

template <class T, class Ref, class Ptr, size_t BufSiz>
inline random_access_iterator_tag
iterator_category(const __deque_iterator<T, Ref, Ptr, BufSiz>&) {
  return random_access_iterator_tag();
}

template <class T, class Ref, class Ptr, size_t BufSiz>
inline T* value_type(const __deque_iterator<T, Ref, Ptr, BufSiz>&) {
  return 0;
}

template <class T, class Ref, class Ptr, size_t BufSiz>
inline ptrdiff_t* distance_type(const __deque_iterator<T, Ref, Ptr, BufSiz>&) {
  return 0;
}

#else /* __STL_NON_TYPE_TMPL_PARAM_BUG */

template <class T, class Ref, class Ptr>
inline random_access_iterator_tag
iterator_category(const __deque_iterator<T, Ref, Ptr>&) {
  return random_access_iterator_tag();
}

template <class T, class Ref, class Ptr>
inline T* value_type(const __deque_iterator<T, Ref, Ptr>&) { return 0; }

template <class T, class Ref, class Ptr>
inline ptrdiff_t* distance_type(const __deque_iterator<T, Ref, Ptr>&) {
  return 0;
}
```

```
#endif /* __STL_NON_TYPE_TMPL_PARAM_BUG */

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

// See __deque_buf_size().  The only reason that the default value is 0
//  is as a workaround for bugs in the way that some compilers handle
//  constant expressions.
template <class T, class Alloc = alloc, size_t BufSiz = 0>
class deque {
public:                     // Basic types
  typedef T value_type;
  typedef value_type* pointer;
  typedef const value_type* const_pointer;
  typedef value_type& reference;
  typedef const value_type& const_reference;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;

public:                     // Iterators
#ifndef __STL_NON_TYPE_TMPL_PARAM_BUG
  typedef __deque_iterator<T, T&, T*, BufSiz>           iterator;
  typedef __deque_iterator<T, const T&, const T&, BufSiz>  const_iterator;
#else /* __STL_NON_TYPE_TMPL_PARAM_BUG */
  typedef __deque_iterator<T, T&, T*>                    iterator;
  typedef __deque_iterator<T, const T&, const T*>        const_iterator;
#endif /* __STL_NON_TYPE_TMPL_PARAM_BUG */

#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION
  typedef reverse_iterator<const_iterator> const_reverse_iterator;
  typedef reverse_iterator<iterator> reverse_iterator;
#else /* __STL_CLASS_PARTIAL_SPECIALIZATION */
  typedef reverse_iterator<const_iterator, value_type, const_reference,
                           difference_type>
          const_reverse_iterator;
  typedef reverse_iterator<iterator, value_type, reference, difference_type>
          reverse_iterator;
#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

protected:                  // Internal typedefs
  typedef pointer* map_pointer;
  typedef simple_alloc<value_type, Alloc> data_allocator;
  typedef simple_alloc<pointer, Alloc> map_allocator;

  static size_type buffer_size() {
    return __deque_buf_size(BufSiz, sizeof(value_type));
  }
  static size_type initial_map_size() { return 8; }
```

```
protected:                  // Data members
  iterator start;
  iterator finish;

  map_pointer map;
  size_type map_size;

public:                     // Basic accessors
  iterator begin() { return start; }
  iterator end() { return finish; }
  const_iterator begin() const { return start; }
  const_iterator end() const { return finish; }

  reverse_iterator rbegin() { return reverse_iterator(finish); }
  reverse_iterator rend() { return reverse_iterator(start); }
  const_reverse_iterator rbegin() const {
    return const_reverse_iterator(finish);
  }
  const_reverse_iterator rend() const {
    return const_reverse_iterator(start);
  }

  reference operator[](size_type n) { return start[difference_type(n)]; }
  const_reference operator[](size_type n) const {
    return start[difference_type(n)];
  }

  reference front() { return *start; }
  reference back() {
    iterator tmp = finish;
    --tmp;
    return *tmp;
  }
  const_reference front() const { return *start; }
  const_reference back() const {
    const_iterator tmp = finish;
    --tmp;
    return *tmp;
  }

  size_type size() const { return finish - start;; }
  size_type max_size() const { return size_type(-1); }
  bool empty() const { return finish == start; }

public:                     // Constructor, destructor.
  deque()
    : start(), finish(), map(0), map_size(0)
  {
    create_map_and_nodes(0);
  }

  deque(const deque& x)
    : start(), finish(), map(0), map_size(0)
  {
    create_map_and_nodes(x.size());
    __STL_TRY {
      uninitialized_copy(x.begin(), x.end(), start);
    }
    __STL_UNWIND(destroy_map_and_nodes());
  }

  deque(size_type n, const value_type& value)
    : start(), finish(), map(0), map_size(0)
  {
    fill_initialize(n, value);
  }

  deque(int n, const value_type& value)
    : start(), finish(), map(0), map_size(0)
  {
    fill_initialize(n, value);
  }

  deque(long n, const value_type& value)
    : start(), finish(), map(0), map_size(0)
  {
    fill_initialize(n, value);
  }

  explicit deque(size_type n)
    : start(), finish(), map(0), map_size(0)
  {
    fill_initialize(n, value_type());
  }

#ifdef __STL_MEMBER_TEMPLATES

  template <class InputIterator>
  deque(InputIterator first, InputIterator last)
    : start(), finish(), map(0), map_size(0)
  {
    range_initialize(first, last, iterator_category(first));
  }

#else /* __STL_MEMBER_TEMPLATES */

  deque(const value_type* first, const value_type* last)
    : start(), finish(), map(0), map_size(0)
  {
    create_map_and_nodes(last - first);
    __STL_TRY {
      uninitialized_copy(first, last, start);
    }
    __STL_UNWIND(destroy_map_and_nodes());
  }

  deque(const_iterator first, const_iterator last)
    : start(), finish(), map(0), map_size(0)
  {
    create_map_and_nodes(last - first);
    __STL_TRY {
      uninitialized_copy(first, last, start);
    }
    __STL_UNWIND(destroy_map_and_nodes());
  }

#endif /* __STL_MEMBER_TEMPLATES */

  ~deque() {
    destroy(start, finish);
    destroy_map_and_nodes();
  }

  deque& operator= (const deque& x) {
    const size_type len = size();
    if (&x != this) {
      if (len >= x.size())
        erase(copy(x.begin(), x.end(), start), finish);
      else {
        const_iterator mid = x.begin() + difference_type(len);
        copy(x.begin(), mid, start);
        insert(finish, mid, x.end());
      }
    }
    return *this;
  }

  void swap(deque& x) {
    __STD::swap(start, x.start);
    __STD::swap(finish, x.finish);
    __STD::swap(map, x.map);
    __STD::swap(map_size, x.map_size);
  }

public:                        // push_* and pop_*

  void push_back(const value_type& t) {
    if (finish.cur != finish.last - 1) {
      construct(finish.cur, t);
      ++finish.cur;
    }
    else
      push_back_aux(t);
  }

  void push_front(const value_type& t) {
    if (start.cur != start.first) {
      construct(start.cur - 1, t);
      --start.cur;
    }
    else
      push_front_aux(t);
  }

  void pop_back() {
    if (finish.cur != finish.first) {
      --finish.cur;
      destroy(finish.cur);
    }
    else
      pop_back_aux();
  }

  void pop_front() {
    if (start.cur != start.last - 1) {
      destroy(start.cur);
      ++start.cur;
    }
    else
      pop_front_aux();
  }

public:                     // Insert

  iterator insert(iterator position, const value_type& x) {
    if (position.cur == start.cur) {
      push_front(x);
      return start;
    }
    else if (position.cur == finish.cur) {
      push_back(x);
      iterator tmp = finish;
      --tmp;
      return tmp;
    }
    else {
      return insert_aux(position, x);
    }
  }

  iterator insert(iterator position) { return insert(position, value_type()); }

  void insert(iterator pos, size_type n, const value_type& x);

  void insert(iterator pos, int n, const value_type& x) {
    insert(pos, (size_type) n, x);
  }
  void insert(iterator pos, long n, const value_type& x) {
    insert(pos, (size_type) n, x);
  }

#ifdef __STL_MEMBER_TEMPLATES

  template <class InputIterator>
  void insert(iterator pos, InputIterator first, InputIterator last) {
    insert(pos, first, last, iterator_category(first));
  }

#else /* __STL_MEMBER_TEMPLATES */

  void insert(iterator pos, const value_type* first, const value_type* last);
  void insert(iterator pos, const_iterator first, const_iterator last);

#endif /* __STL_MEMBER_TEMPLATES */

  void resize(size_type new_size, const value_type& x) {
    const size_type len = size();
    if (new_size < len)
      erase(start + new_size, finish);
    else
      insert(finish, new_size - len, x);
  }

  void resize(size_type new_size) { resize(new_size, value_type()); }

public:                     // Erase
  iterator erase(iterator pos) {
    iterator next = pos;
    ++next;
    difference_type index = pos - start;
    if (index < (size() >> 1)) {
      copy_backward(start, pos, next);
      pop_front();
    }
    else {
     copy(next, finish, pos);
     pop_back();
   }
   return start + index;
  }

  iterator erase(iterator first, iterator last);
  void clear();

protected:                    // Internal construction/destruction

  void create_map_and_nodes(size_type num_elements);
  void destroy_map_and_nodes();
  void fill_initialize(size_type n, const value_type& value);

#ifdef __STL_MEMBER_TEMPLATES

  template <class InputIterator>
  void range_initialize(InputIterator first, InputIterator last,
                    input_iterator_tag);

  template <class ForwardIterator>
  void range_initialize(ForwardIterator first, ForwardIterator last,
                    forward_iterator_tag);

#endif /* __STL_MEMBER_TEMPLATES */

protected:                    // Internal push_* and pop_*

  void push_back_aux(const value_type& t);
  void push_front_aux(const value_type& t);
  void pop_back_aux();
  void pop_front_aux();

protected:                    // Internal insert functions

#ifdef __STL_MEMBER_TEMPLATES

  template <class InputIterator>
  void insert(iterator pos, InputIterator first, InputIterator last,
            input_iterator_tag);

  template <class ForwardIterator>
  void insert(iterator pos, ForwardIterator first, ForwardIterator last,
            forward_iterator_tag);

#endif /* __STL_MEMBER_TEMPLATES */

  iterator insert_aux(iterator pos, const value_type& x);
  void insert_aux(iterator pos, size_type n, const value_type& x);

#ifdef __STL_MEMBER_TEMPLATES

  template <class ForwardIterator>
  void insert_aux(iterator pos, ForwardIterator first, ForwardIterator last,
                size_type n);

#else /* __STL_MEMBER_TEMPLATES */

  void insert_aux(iterator pos,
                const value_type* first, const value_type* last,
                size_type n);

  void insert_aux(iterator pos, const_iterator first, const_iterator last,
                size_type n);

#endif /* __STL_MEMBER_TEMPLATES */

  iterator reserve_elements_at_front(size_type n) {
    size_type vacancies = start.cur - start.first;
    if (n > vacancies)
      new_elements_at_front(n - vacancies);
    return start - difference_type(n);
  }

  iterator reserve_elements_at_back(size_type n) {
    size_type vacancies = (finish.last - finish.cur) - 1;
    if (n > vacancies)
      new_elements_at_back(n - vacancies);
    return finish + difference_type(n);
  }

  void new_elements_at_front(size_type new_elements);
  void new_elements_at_back(size_type new_elements);

  void destroy_nodes_at_front(iterator before_start);
  void destroy_nodes_at_back(iterator after_finish);

protected:                   // Allocation of map and nodes

  // Makes sure the map has space for new nodes.  Does not actually
  //  add the nodes.  Can invalidate map pointers.  (And consequently,
  //  deque iterators.)

  void reserve_map_at_back (size_type nodes_to_add = 1) {
    if (nodes_to_add + 1 > map_size - (finish.node - map))
      reallocate_map(nodes_to_add, false);
  }
  void reserve_map_at_front (size_type nodes_to_add = 1) {
    if (nodes_to_add > start.node - map)
      reallocate_map(nodes_to_add, true);
  }

  void reallocate_map(size_type nodes_to_add, bool add_at_front);

  pointer allocate_node() { return data_allocator::allocate(buffer_size()); }
  void deallocate_node(pointer n) {
    data_allocator::deallocate(n, buffer_size());
  }

#ifdef __STL_NON_TYPE_TMPL_PARAM_BUG
public:
  bool operator==(const deque<T, Alloc, 0>& x) const {
    return size() == x.size() && equal(begin(), end(), x.begin());
  }
  bool operator!=(const deque<T, Alloc, 0>& x) const {
    return size() != x.size() || !equal(begin(), end(), x.begin());
  }
  bool operator<(const deque<T, Alloc, 0>& x) const {
    return lexicographical_compare(begin(), end(), x.begin(), x.end());
  }
#endif /* __STL_NON_TYPE_TMPL_PARAM_BUG */
};

// Non-inline member functions

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::insert(iterator pos,
                                      size_type n, const value_type& x) {
  if (pos.cur == start.cur) {
    iterator new_start = reserve_elements_at_front(n);
    uninitialized_fill(new_start, start, x);
    start = new_start;
  }
  else if (pos.cur == finish.cur) {
    iterator new_finish = reserve_elements_at_back(n);
    uninitialized_fill(finish, new_finish, x);
    finish = new_finish;
  }
  else
    insert_aux(pos, n, x);
}

#ifndef __STL_MEMBER_TEMPLATES

template <class T, class Alloc, size_t BufSize>
```

```
void deque<T, Alloc, BufSize>::insert(iterator pos,
                                      const value_type* first,
                                      const value_type* last) {
  size_type n = last - first;
  if (pos.cur == start.cur) {
    iterator new_start = reserve_elements_at_front(n);
    __STL_TRY {
      uninitialized_copy(first, last, new_start);
      start = new_start;
    }
    __STL_UNWIND(destroy_nodes_at_front(new_start));
  }
  else if (pos.cur == finish.cur) {
    iterator new_finish = reserve_elements_at_back(n);
    __STL_TRY {
      uninitialized_copy(first, last, finish);
      finish = new_finish;
    }
    __STL_UNWIND(destroy_nodes_at_back(new_finish));
  }
  else
    insert_aux(pos, first, last, n);
}

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::insert(iterator pos,
                                      const_iterator first,
                                      const_iterator last)
{
  size_type n = last - first;
  if (pos.cur == start.cur) {
    iterator new_start = reserve_elements_at_front(n);
    __STL_TRY {
      uninitialized_copy(first, last, new_start);
      start = new_start;
    }
    __STL_UNWIND(destroy_nodes_at_front(new_start));
  }
  else if (pos.cur == finish.cur) {
    iterator new_finish = reserve_elements_at_back(n);
    __STL_TRY {
      uninitialized_copy(first, last, finish);
      finish = new_finish;
    }
    __STL_UNWIND(destroy_nodes_at_back(new_finish));
  }
  else
    insert_aux(pos, first, last, n);
}
```

```
#endif /* __STL_MEMBER_TEMPLATES */

template <class T, class Alloc, size_t BufSize>
deque<T, Alloc, BufSize>::iterator
deque<T, Alloc, BufSize>::erase(iterator first, iterator last) {
  if (first == start && last == finish) {
    clear();
    return finish;
  }
  else {
    difference_type n = last - first;
    difference_type elems_before = first - start;
    if (elems_before < (size() - n) / 2) {
      copy_backward(start, first, last);
      iterator new_start = start + n;
      destroy(start, new_start);
      for (map_pointer cur = start.node; cur < new_start.node; ++cur)
        data_allocator::deallocate(*cur, buffer_size());
      start = new_start;
    }
    else {
      copy(last, finish, first);
      iterator new_finish = finish - n;
      destroy(new_finish, finish);
      for (map_pointer cur = new_finish.node + 1; cur <= finish.node; ++cur)
        data_allocator::deallocate(*cur, buffer_size());
      finish = new_finish;
    }
    return start + elems_before;
  }
}

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::clear() {
  for (map_pointer node = start.node + 1; node < finish.node; ++node) {
    destroy(*node, *node + buffer_size());
    data_allocator::deallocate(*node, buffer_size());
  }

  if (start.node != finish.node) {
    destroy(start.cur, start.last);
    destroy(finish.first, finish.cur);
    data_allocator::deallocate(finish.first, buffer_size());
  }
  else
    destroy(start.cur, finish.cur);

  finish = start;
```

```
}

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::create_map_and_nodes(size_type num_elements) {
  size_type num_nodes = num_elements / buffer_size() + 1;

  map_size = max(initial_map_size(), num_nodes + 2);
  map = map_allocator::allocate(map_size);

  map_pointer nstart = map + (map_size - num_nodes) / 2;
  map_pointer nfinish = nstart + num_nodes - 1;

  map_pointer cur;
  __STL_TRY {
    for (cur = nstart; cur <= nfinish; ++cur)
      *cur = allocate_node();
  }
#     ifdef  __STL_USE_EXCEPTIONS
  catch(...) {
    for (map_pointer n = nstart; n < cur; ++n)
      deallocate_node(*n);
    map_allocator::deallocate(map, map_size);
    throw;
  }
#     endif /* __STL_USE_EXCEPTIONS */

  start.set_node(nstart);
  finish.set_node(nfinish);
  start.cur = start.first;
  finish.cur = finish.first + num_elements % buffer_size();
}

// This is only used as a cleanup function in catch clauses.
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::destroy_map_and_nodes() {
  for (map_pointer cur = start.node; cur <= finish.node; ++cur)
    deallocate_node(*cur);
  map_allocator::deallocate(map, map_size);
}


template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::fill_initialize(size_type n,
                                               const value_type& value) {
  create_map_and_nodes(n);
  map_pointer cur;
  __STL_TRY {
    for (cur = start.node; cur < finish.node; ++cur)
      uninitialized_fill(*cur, *cur + buffer_size(), value);
    uninitialized_fill(finish.first, finish.cur, value);
  }
#       ifdef __STL_USE_EXCEPTIONS
  catch(...) {
    for (map_pointer n = start.node; n < cur; ++n)
      destroy(*n, *n + buffer_size());
    destroy_map_and_nodes();
    throw;
  }
#       endif /* __STL_USE_EXCEPTIONS */
}

#ifdef __STL_MEMBER_TEMPLATES

template <class T, class Alloc, size_t BufSize>
template <class InputIterator>
void deque<T, Alloc, BufSize>::range_initialize(InputIterator first,
                                                InputIterator last,
                                                input_iterator_tag) {
  create_map_and_nodes(0);
  for ( ; first != last; ++first)
    push_back(*first);
}

template <class T, class Alloc, size_t BufSize>
template <class ForwardIterator>
void deque<T, Alloc, BufSize>::range_initialize(ForwardIterator first,
                                                ForwardIterator last,
                                                forward_iterator_tag) {
  size_type n = 0;
  distance(first, last, n);
  create_map_and_nodes(n);
  __STL_TRY {
    uninitialized_copy(first, last, start);
  }
  __STL_UNWIND(destroy_map_and_nodes());
}

#endif /* __STL_MEMBER_TEMPLATES */

// Called only if finish.cur == finish.last - 1.
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::push_back_aux(const value_type& t) {
  value_type t_copy = t;
  reserve_map_at_back();
  *(finish.node + 1) = allocate_node();
  __STL_TRY {
    construct(finish.cur, t_copy);
    finish.set_node(finish.node + 1);
    finish.cur = finish.first;
  }
  __STL_UNWIND(deallocate_node(*(finish.node + 1)));
}

// Called only if start.cur == start.first.
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::push_front_aux(const value_type& t) {
  value_type t_copy = t;
  reserve_map_at_front();
  *(start.node - 1) = allocate_node();
  __STL_TRY {
    start.set_node(start.node - 1);
    start.cur = start.last - 1;
    construct(start.cur, t_copy);
  }
#    ifdef __STL_USE_EXCEPTIONS
  catch(...) {
    start.set_node(start.node + 1);
    start.cur = start.first;
    deallocate_node(*(start.node - 1));
    throw;
  }
#    endif /* __STL_USE_EXCEPTIONS */
}

// Called only if finish.cur == finish.first.
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>:: pop_back_aux() {
  deallocate_node(finish.first);
  finish.set_node(finish.node - 1);
  finish.cur = finish.last - 1;
  destroy(finish.cur);
}

// Called only if start.cur == start.last - 1.  Note that if the deque
//  has at least one element (a necessary precondition for this member
//  function), and if start.cur == start.last, then the deque must have
//  at least two nodes.
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::pop_front_aux() {
  destroy(start.cur);
  deallocate_node(start.first);
  start.set_node(start.node + 1);
  start.cur = start.first;
}

#ifdef __STL_MEMBER_TEMPLATES
```

```
template <class T, class Alloc, size_t BufSize>
template <class InputIterator>
void deque<T, Alloc, BufSize>::insert(iterator pos,
                                InputIterator first, InputIterator last,
                                input_iterator_tag) {
  copy(first, last, inserter(*this, pos));
}

template <class T, class Alloc, size_t BufSize>
template <class ForwardIterator>
void deque<T, Alloc, BufSize>::insert(iterator pos,
                                ForwardIterator first,
                                ForwardIterator last,
                                forward_iterator_tag) {
  size_type n = 0;
  distance(first, last, n);
  if (pos.cur == start.cur) {
    iterator new_start = reserve_elements_at_front(n);
    __STL_TRY {
      uninitialized_copy(first, last, new_start);
      start = new_start;
    }
    __STL_UNWIND(destroy_nodes_at_front(new_start));
  }
  else if (pos.cur == finish.cur) {
    iterator new_finish = reserve_elements_at_back(n);
    __STL_TRY {
      uninitialized_copy(first, last, finish);
      finish = new_finish;
    }
    __STL_UNWIND(destroy_nodes_at_back(new_finish));
  }
  else
    insert_aux(pos, first, last, n);
}

#endif /* __STL_MEMBER_TEMPLATES */

template <class T, class Alloc, size_t BufSize>
typename deque<T, Alloc, BufSize>::iterator
deque<T, Alloc, BufSize>::insert_aux(iterator pos, const value_type& x) {
  difference_type index = pos - start;
  value_type x_copy = x;
  if (index < size() / 2) {
    push_front(front());
    iterator front1 = start;
    ++front1;
    iterator front2 = front1;
    ++front2;
    pos = start + index;
    iterator pos1 = pos;
    ++pos1;
    copy(front2, pos1, front1);
  }
  else {
    push_back(back());
    iterator back1 = finish;
    --back1;
    iterator back2 = back1;
    --back2;
    pos = start + index;
    copy_backward(pos, back2, back1);
  }
  *pos = x_copy;
  return pos;
}

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::insert_aux(iterator pos,
                                          size_type n, const value_type& x) {
  const difference_type elems_before = pos - start;
  size_type length = size();
  value_type x_copy = x;
  if (elems_before < length / 2) {
    iterator new_start = reserve_elements_at_front(n);
    iterator old_start = start;
    pos = start + elems_before;
    __STL_TRY {
      if (elems_before >= difference_type(n)) {
        iterator start_n = start + difference_type(n);
        uninitialized_copy(start, start_n, new_start);
        start = new_start;
        copy(start_n, pos, old_start);
        fill(pos - difference_type(n), pos, x_copy);
      }
      else {
        __uninitialized_copy_fill(start, pos, new_start, start, x_copy);
        start = new_start;
        fill(old_start, pos, x_copy);
      }
    }
    __STL_UNWIND(destroy_nodes_at_front(new_start));
  }
  else {
    iterator new_finish = reserve_elements_at_back(n);
    iterator old_finish = finish;
    const difference_type elems_after = difference_type(length) - elems_before;
    pos = finish - elems_after;
    __STL_TRY {
      if (elems_after > difference_type(n)) {
        iterator finish_n = finish - difference_type(n);
        uninitialized_copy(finish_n, finish, finish);
        finish = new_finish;
        copy_backward(pos, finish_n, old_finish);
        fill(pos, pos + difference_type(n), x_copy);
      }
      else {
        __uninitialized_fill_copy(finish, pos + difference_type(n),
                                  x_copy,
                                  pos, finish);
        finish = new_finish;
        fill(pos, old_finish, x_copy);
      }
    }
    __STL_UNWIND(destroy_nodes_at_back(new_finish));
  }
}

#ifdef __STL_MEMBER_TEMPLATES

template <class T, class Alloc, size_t BufSize>
template <class ForwardIterator>
void deque<T, Alloc, BufSize>::insert_aux(iterator pos,
                                          ForwardIterator first,
                                          ForwardIterator last,
                                          size_type n)
{
  const difference_type elems_before = pos - start;
  size_type length = size();
  if (elems_before < length / 2) {
    iterator new_start = reserve_elements_at_front(n);
    iterator old_start = start;
    pos = start + elems_before;
    __STL_TRY {
      if (elems_before >= difference_type(n)) {
        iterator start_n = start + difference_type(n);
        uninitialized_copy(start, start_n, new_start);
        start = new_start;
        copy(start_n, pos, old_start);
        copy(first, last, pos - difference_type(n));
      }
      else {
        ForwardIterator mid = first;
        advance(mid, difference_type(n) - elems_before);
        __uninitialized_copy_copy(start, pos, first, mid, new_start);
        start = new_start;
        copy(mid, last, old_start);
      }
    }
    __STL_UNWIND(destroy_nodes_at_front(new_start));
  }
  else {
    iterator new_finish = reserve_elements_at_back(n);
    iterator old_finish = finish;
    const difference_type elems_after = difference_type(length) - elems_before;
    pos = finish - elems_after;
    __STL_TRY {
      if (elems_after > difference_type(n)) {
        iterator finish_n = finish - difference_type(n);
        uninitialized_copy(finish_n, finish, finish);
        finish = new_finish;
        copy_backward(pos, finish_n, old_finish);
        copy(first, last, pos);
      }
      else {
        ForwardIterator mid = first;
        advance(mid, elems_after);
        __uninitialized_copy_copy(mid, last, pos, finish, finish);
        finish = new_finish;
        copy(first, mid, pos);
      }
    }
    __STL_UNWIND(destroy_nodes_at_back(new_finish));
  }
}

#else /* __STL_MEMBER_TEMPLATES */

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::insert_aux(iterator pos,
                                          const value_type* first,
                                          const value_type* last,
                                          size_type n)
{
  const difference_type elems_before = pos - start;
  size_type length = size();
  if (elems_before < length / 2) {
    iterator new_start = reserve_elements_at_front(n);
    iterator old_start = start;
    pos = start + elems_before;
    __STL_TRY {
      if (elems_before >= difference_type(n)) {
        iterator start_n = start + difference_type(n);
        uninitialized_copy(start, start_n, new_start);
        start = new_start;
        copy(start_n, pos, old_start);
        copy(first, last, pos - difference_type(n));
      }
      else {
        const value_type* mid = first + (difference_type(n) - elems_before);
        __uninitialized_copy_copy(start, pos, first, mid, new_start);
        start = new_start;
        copy(mid, last, old_start);
      }
    }
    __STL_UNWIND(destroy_nodes_at_front(new_start));
  }
  else {
    iterator new_finish = reserve_elements_at_back(n);
    iterator old_finish = finish;
    const difference_type elems_after = difference_type(length) - elems_before;
    pos = finish - elems_after;
    __STL_TRY {
      if (elems_after > difference_type(n)) {
        iterator finish_n = finish - difference_type(n);
        uninitialized_copy(finish_n, finish, finish);
        finish = new_finish;
        copy_backward(pos, finish_n, old_finish);
        copy(first, last, pos);
      }
      else {
        const value_type* mid = first + elems_after;
        __uninitialized_copy_copy(mid, last, pos, finish, finish);
        finish = new_finish;
        copy(first, mid, pos);
      }
    }
    __STL_UNWIND(destroy_nodes_at_back(new_finish));
  }
}

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::insert_aux(iterator pos,
                                   const_iterator first,
                                   const_iterator last,
                                   size_type n)
{
  const difference_type elems_before = pos - start;
  size_type length = size();
  if (elems_before < length / 2) {
    iterator new_start = reserve_elements_at_front(n);
    iterator old_start = start;
    pos = start + elems_before;
    __STL_TRY {
      if (elems_before >= n) {
        iterator start_n = start + n;
        uninitialized_copy(start, start_n, new_start);
        start = new_start;
        copy(start_n, pos, old_start);
        copy(first, last, pos - difference_type(n));
      }
      else {
        const_iterator mid = first + (n - elems_before);
        __uninitialized_copy_copy(start, pos, first, mid, new_start);
        start = new_start;
        copy(mid, last, old_start);
      }
    }
    __STL_UNWIND(destroy_nodes_at_front(new_start));
  }
  else {
    iterator new_finish = reserve_elements_at_back(n);
    iterator old_finish = finish;
    const difference_type elems_after = length - elems_before;
    pos = finish - elems_after;
    __STL_TRY {
      if (elems_after > n) {
        iterator finish_n = finish - difference_type(n);
        uninitialized_copy(finish_n, finish, finish);
        finish = new_finish;
        copy_backward(pos, finish_n, old_finish);
        copy(first, last, pos);
      }
      else {
        const_iterator mid = first + elems_after;
        __uninitialized_copy_copy(mid, last, pos, finish, finish);
        finish = new_finish;
        copy(first, mid, pos);
      }
    }
    __STL_UNWIND(destroy_nodes_at_back(new_finish));
  }
}

#endif /* __STL_MEMBER_TEMPLATES */

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::new_elements_at_front(size_type new_elements) {
  size_type new_nodes = (new_elements + buffer_size() - 1) / buffer_size();
  reserve_map_at_front(new_nodes);
  size_type i;
  __STL_TRY {
    for (i = 1; i <= new_nodes; ++i)
      *(start.node - i) = allocate_node();
  }
#      ifdef __STL_USE_EXCEPTIONS
  catch(...) {
    for (size_type j = 1; j < i; ++j)
      deallocate_node(*(start.node - j));
    throw;
  }
#      endif /* __STL_USE_EXCEPTIONS */
}

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::new_elements_at_back(size_type new_elements) {
  size_type new_nodes = (new_elements + buffer_size() - 1) / buffer_size();
  reserve_map_at_back(new_nodes);
  size_type i;
  __STL_TRY {
    for (i = 1; i <= new_nodes; ++i)
      *(finish.node + i) = allocate_node();
  }
#      ifdef __STL_USE_EXCEPTIONS
  catch(...) {
    for (size_type j = 1; j < i; ++j)
      deallocate_node(*(finish.node + j));
    throw;
  }
#      endif /* __STL_USE_EXCEPTIONS */
}

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::destroy_nodes_at_front(iterator before_start) {
  for (map_pointer n = before_start.node; n < start.node; ++n)
    deallocate_node(*n);
}

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::destroy_nodes_at_back(iterator after_finish) {
  for (map_pointer n = after_finish.node; n > finish.node; --n)
    deallocate_node(*n);
}

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::reallocate_map(size_type nodes_to_add,
                                      bool add_at_front) {
  size_type old_num_nodes = finish.node - start.node + 1;
  size_type new_num_nodes = old_num_nodes + nodes_to_add;

  map_pointer new_nstart;
  if (map_size > 2 * new_num_nodes) {
    new_nstart = map + (map_size - new_num_nodes) / 2
```

```
                 + (add_at_front ? nodes_to_add : 0);
   if (new_nstart < start.node)
     copy(start.node, finish.node + 1, new_nstart);
   else
     copy_backward(start.node, finish.node + 1, new_nstart + old_num_nodes);
  }
  else {
   size_type new_map_size = map_size + max(map_size, nodes_to_add) + 2;

   map_pointer new_map = map_allocator::allocate(new_map_size);
   new_nstart = new_map + (new_map_size - new_num_nodes) / 2
                    + (add_at_front ? nodes_to_add : 0);
   copy(start.node, finish.node + 1, new_nstart);
   map_allocator::deallocate(map, map_size);

   map = new_map;
   map_size = new_map_size;
  }

  start.set_node(new_nstart);
  finish.set_node(new_nstart + old_num_nodes - 1);
}


// Nonmember functions.

#ifndef __STL_NON_TYPE_TMPL_PARAM_BUG

template <class T, class Alloc, size_t BufSiz>
bool operator==(const deque<T, Alloc, BufSiz>& x,
             const deque<T, Alloc, BufSiz>& y) {
  return x.size() == y.size() && equal(x.begin(), x.end(), y.begin());
}

template <class T, class Alloc, size_t BufSiz>
bool operator<(const deque<T, Alloc, BufSiz>& x,
            const deque<T, Alloc, BufSiz>& y) {
  return lexicographical_compare(x.begin(), x.end(), y.begin(), y.end());
}

#endif /* __STL_NON_TYPE_TMPL_PARAM_BUG */

#if defined(__STL_FUNCTION_TMPL_PARTIAL_ORDER) && \
   !defined(__STL_NON_TYPE_TMPL_PARAM_BUG)

template <class T, class Alloc, size_t BufSiz>
inline void swap(deque<T, Alloc, BufSiz>& x, deque<T, Alloc, BufSiz>& y) {
  x.swap(y);
}
```

```
#endif

#if defined(__sgi) && !defined(__GNUC__) && (_MIPS_SIM != _MIPS_SIM_ABI32)
#pragma reset woff 1174
#endif

__STL_END_NAMESPACE

#endif /* __SGI_STL_INTERNAL_DEQUE_H */

// Local Variables:
// mode:C++
// End:
```