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 的恆長特性（invariants）：
 *  對於任何 nonsingular iterator I：
 *    i.node 是 map array 中的某個元素的位址。
 *      i.node 所指內容則是一個指標，指向某個節點（緩衝區）的頭。
 *    i.first == *(i.node)
 *    i.last  == i.first + node_size（也就是 buffer_size()）
 *    i.cur 是一個指標，指向範圍 [i.first, i.last) 之間。注意：
 *      這意味 i.cur 永遠是一個 dereferenceable pointer,
 *      縱使 i 是一個 past-the-end iterator.
 *  Start 和 Finish 總是 nonsingular iterators。注意：這意味
 *    empty deque 一定會有一個node，而一個具有N個元素的deque，
 *    （N 表示緩衝區大小），一定會有兩個nodes。
 *  對於start.node 和finish.node 以外的每一個node，其中的每一個元素
 *    都是一個經過初始化的物件。如果 start.node == finish.node，
 *    那麼 [start.cur, finish.cur) 都是經過初始化的物件，而該範圍以外
```

```
 *    元素則是未經初始化的空間。否則，[start.cur, start.last) 和
 *    [finish.first, finish.cur) 是經過初始化的物件，而
 *    [start.first, start.cur) 和 [finish.cur, finish.last)
 *    則是未經初始化的空間
 *  [map, map + map_size) 是一個有效的，non-empty 的範圍。
 *  [start.node, finish.node] 是一個有效的範圍，內含於
 *    [map, map + map_size) 之內。
 *  範圍 [map, map + map_size) 內的任何一個指標會指向一個經過配置的
 *    node 一 若且唯若該指標在範圍 [start.node, finish.node] 之內。
 */

/*
 * 在前一版的deque中，node_size 由編譯器定死。這個版本允許使用者選擇節點
*（node）的大小。Deque有三個 template參數，其中第三個是一個型別為 size_t
 * 的數值，代表每個節點（node）內含的元素個數。如果第三個 template 參數為0
 * （那是預設值），deque 就使用預設的節點大小。
 *
 * 使用不同的節點大小的唯一理由是，或許你的程式需要不同的效率並願意付出其他方
 * 面的代價。例如，假設你的程式內含許多deques，每一個都只內含一些元素，那麼
 * 你可以使用較小的 nodes 來節省記憶體（或許會因此犧牲速度）。
 *
 * 不幸的是，某些編譯器面對 non-type template 參數會有問題。stl_config.h
之
 * 中為此定義了一個 __STL_NON_TYPE_TMPL_PARAM_BUG。如果你的編譯器正是如
 * 此，你就無法使用不同的節點大小，你必須使用預設大小。
*/

__STL_BEGIN_NAMESPACE

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

// 注意：下面這個函式是七拼八湊的產品，只是為了閃避數家編譯器在處理常數算式
// （constant expressions）時的臭蟲。
// 如果 n 不為 0，傳回 n，表示 buffer size 由使用者自定。
// 如果 n 為 0，表示buffer size 使用預設值，那麼
//   如果 sz（元素大小，sizeof(value_type)）小於 512，傳回 512/sz，
//   如果 sz 不小於 512，傳回 1。
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 {  // 未繼承 std::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 {  // 未繼承 std::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

  // 未繼承 std::iterator，所以必須自行撰寫五個必要的迭代器相應型別
  typedef random_access_iterator_tag iterator_category; // (1)
  typedef T value_type;                    // (2)
  typedef Ptr pointer;                     // (3)
  typedef Ref reference;                   // (4)
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;       // (5)
  typedef T** map_pointer;

  typedef __deque_iterator self;

  // 保持與容器的聯結
  T* cur;    // 此迭代器所指之緩衝區中的現行（current）元素
  T* first;  // 此迭代器所指之緩衝區的頭
  T* last;   // 此迭代器所指之緩衝區的尾（含備用空間）
  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) {}


// 以下各個多載化運算子是 __deque_iterator<> 成功運作的關鍵。

  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);
  }

  // 參考 More Effective C++, item6: Distinguish between prefix and
  // postfix forms of increment and decrement operators.
  self& operator++() {
```

```
    ++cur;                  // 切換至下一個元素。
    if (cur == last) {      // 如果已達所在緩衝區的尾端，
      set_node(node + 1);   // 就切換至下一個節點（亦即緩衝區）
      cur = first;          //   的第一個元素。
    }
    return *this;
  }
  self operator++(int)  {
    self tmp = *this;
    ++*this;
    return tmp;
  }
  self& operator--() {
    if (cur == first) {     // 如果已達所在緩衝區的頭端，
      set_node(node - 1);   // 就切換至前一個節點（亦即緩衝區）
      cur = last;           //   的最後一個元素。
    }
    --cur;                  // 切換至前一個元素。
    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()))
      // 目標位置在同一緩衝區內
      cur += n;
    else {
      // 目標位置不在同一緩衝區內
      difference_type node_offset =
        offset > 0 ? offset / difference_type(buffer_size())
                   : -difference_type((-offset - 1) / buffer_size()) - 1;
      // 切換至正確的節點（亦即緩衝區）
      set_node(node + node_offset);
      // 切換至正確的元素
      cur = first + (offset - node_offset * difference_type(buffer_size()));
    }
    return *this;
  }

  // 參考 More Effective C++, item22: Consider using op= instead of
  // stand-alone op.
  self operator+(difference_type n) const {
    self tmp = *this;
    return tmp += n; // 喚起operator+=
```

```
  }

  self& operator-=(difference_type n) { return *this += -n; }
  // 以上利用operator+= 來完成 operator-=

  // 參考 More Effective C++, item22: Consider using op= instead of
  // stand-alone op.
  self operator-(difference_type n) const {
    self tmp = *this;
    return tmp -= n; // 喚起operator-=
  }

  reference operator[](difference_type n) const { return *(*this + n); }
  // 以上喚起operator*, operator+

  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
// 編譯器不支援 partial specialization 時，才需以下定義
#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 */
// 編譯器不支援 partial specialization 時，才需以上定義
#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

// 見 __deque_buf_size()。BufSize 預設值為 0 的唯一理由是為了閃避某些
//  編譯器在處理常數算式（constant expressions）時的臭蟲。
// 預設使用 alloc 為配置器
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
  // 元素的指標的指標（pointer of pointer of T）
  typedef pointer* map_pointer;
  // 專屬之空間配置器，每次配置一個元素大小
  typedef simple_alloc<value_type, Alloc> data_allocator;
  // 專屬之空間配置器，每次配置一個指標大小
  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;      // 表現第一個節點。
  iterator finish;     // 表現最後一個節點。

  map_pointer map;     // 指向map，map是塊連續空間，
                       // 其內的每個元素都是一個指標（稱為節點），指向一塊緩衝區。
  size_type map_size;  // map內可容納多少指標。

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)]; // 喚起 __deque_iterator<>::operator[]
  }
  const_reference operator[](size_type n) const {
    return start[difference_type(n)];
  }
```

```
  reference front() { return *start; } // 喚起 __deque_iterator<>::operator*
  reference back() {
    iterator tmp = finish;
    --tmp;    // 喚起 __deque_iterator<>::operator--
    return *tmp;  // 喚起 __deque_iterator<>::operator*
    // 以上三行何不改為：return *(finish-1);
    // 因為 __deque_iterator<> 沒有為 (finish-1) 定義運算子。待查!
  }
  const_reference front() const { return *start; }
  const_reference back() const {
    const_iterator tmp = finish;
    --tmp;
    return *tmp;
  }

  // 下行最後有兩個 ';'，雖奇怪但合乎語法。
  size_type size() const { return finish - start;; }
  // 以上喚起iterator::operator-
  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)
    // 以上 start() 和 finish() 喚起 iterator（亦即 __deque_iterator）
    // 的 default ctor，於是令其 cur, first, last, node 皆為 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) {
      // 最後緩衝區尚有一個以上的備用空間
      construct(finish.cur, t); // 直接在備用空間上建構元素
      ++finish.cur;    // 調整最後緩衝區的使用狀態
    }
    else  // 最後緩衝區已無（或只剩一個）元素備用空間。
      push_back_aux(t);
  }

  void push_front(const value_type& t) {
    if (start.cur != start.first) {  // 第一緩衝區尚有備用空間
      construct(start.cur - 1, t);   // 直接在備用空間上建構元素
      --start.cur;     // 調整第一緩衝區的使用狀態
    }
    else // 第一緩衝區已無備用空間
      push_front_aux(t);
  }

  void pop_back() {
    if (finish.cur != finish.first) {
```

```
      // 最後緩衝區有一個（或更多）元素

      --finish.cur;          // 調整指標，相當於排除了最後元素
      destroy(finish.cur);  // 將最後元素解構
    }
    else
      // 最後緩衝區沒有任何元素
      pop_back_aux();        // 這裡將進行緩衝區的釋放工作
  }

  void pop_front() {
    if (start.cur != start.last - 1) {
      // 第一緩衝區有一個（或更多）元素
      destroy(start.cur);   // 將第一元素解構
      ++start.cur;          // 調整指標，相當於排除了第一元素
    }
    else
      // 第一緩衝區僅有一個元素
      pop_front_aux();      // 這裡將進行緩衝區的釋放工作
  }

public:                     // Insert

  // 在position 處安插一個元素，其值為 x
  iterator insert(iterator position, const value_type& x) {
    if (position.cur == start.cur) { // 如果安插點是deque 最前端
      push_front(x);                 // 交給push_front 去做
      return start;
    }
    else if (position.cur == finish.cur) { // 如果安插點是deque 最尾端
      push_back(x);                         // 交給push_back 去做
      iterator tmp = finish;
      --tmp;
      return tmp;
    }
    else {
      return insert_aux(position, x);     // 交給 insert_aux 去做
    }
  }

  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
  // 清除 pos 所指的元素。pos 為清除點。
  iterator erase(iterator pos) {
    iterator next = pos;
    ++next;
    difference_type index = pos - start;  // 清除點之前的元素個數
    if (index < (size() >> 1)) {          // 如果清除點之前的元素比較少，
      copy_backward(start, pos, next);    // 就搬移清除點之前的元素
      pop_front();                // 搬移完畢，最前一個元素贅餘，去除之
    }
    else {                        // 清除點之後的元素比較少，
      copy(next, finish, pos);    // 就搬移清除點之後的元素
      pop_back();                 // 搬移完畢，最後一個元素贅餘，去除之
    }
    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))
      // 如果 map 尾端的節點備用空間不足
      // 符合以上條件則必須重換一個map（配置更大的，拷貝原來的，釋放原來的）
      reallocate_map(nodes_to_add, false);
  }

  void reserve_map_at_front (size_type nodes_to_add = 1) {
    if (nodes_to_add > start.node - map)
      // 如果 map 前端的節點備用空間不足
      // 符合以上條件則必須重換一個map（配置更大的，拷貝原來的，釋放原來的）
      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) { // 如果清除區間就是整個 deque
    clear();                              // 直接呼叫 clear() 即可
```

```
    return finish;
  }
  else {
    difference_type n = last - first;              // 清除區間的長度
    difference_type elems_before = first - start;  // 清除區間前方的元素個數
    if (elems_before < (size() - n) / 2) {         // 如果前方的元素比較少，
      copy_backward(start, first, last);       // 向後搬移前方元素（覆蓋清除區間）
      iterator new_start = start + n;          // 標記 deque 的新起點
      destroy(start, new_start);               // 搬移完畢，將贅餘的元素解構
      // 以下將贅餘的緩衝區釋放
      for (map_pointer cur = start.node; cur < new_start.node; ++cur)
        data_allocator::deallocate(*cur, buffer_size());
      start = new_start;    // 設定 deque 的新起點
    }
    else {    // 如果清除區間後方的元素比較少
      copy(last, finish, first);           // 向前搬移後方元素（覆蓋清除區間）
      iterator new_finish = finish - n;    // 標記 deque 的新尾點
      destroy(new_finish, finish);         // 搬移完畢，將贅餘的元素解構
      // 以下將贅餘的緩衝區釋放
      for (map_pointer cur = new_finish.node + 1; cur <= finish.node; ++cur)
        data_allocator::deallocate(*cur, buffer_size());
      finish = new_finish;  // 設定 deque 的新尾點
    }
    return start + elems_before;
  }
}

// 注意，最終需要保留一個緩衝區。這是deque 的策略，也是deque 的初始狀態。
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::clear() {
  // 以下針對頭尾以外的每一個緩衝區（它們一定都是飽滿的）
  for (map_pointer node = start.node + 1; node < finish.node; ++node) {
    // 將緩衝區內的所有元素解構。注意，呼叫的是destroy() 第二版本，見 2.2.3 節
    destroy(*node, *node + buffer_size());
    // 釋放緩衝區記憶體
    data_allocator::deallocate(*node, buffer_size());
  }

  if (start.node != finish.node) {   // 至少有頭尾兩個緩衝區
    destroy(start.cur, start.last);  // 將頭緩衝區的目前所有元素解構
    destroy(finish.first, finish.cur); // 將尾緩衝區的目前所有元素解構
    // 以下釋放尾緩衝區。注意，頭緩衝區保留。
    data_allocator::deallocate(finish.first, buffer_size());
  }
  else  // 只有一個緩衝區
    destroy(start.cur, finish.cur);  // 將此唯一緩衝區內的所有元素解構
    // 注意，並不釋放緩衝區空間。這唯一的緩衝區將保留。

  finish = start; // 調整狀態
```

```
}

template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::create_map_and_nodes(size_type num_elements)
{
  // 需要節點數=(元素個數/每個緩衝區可容納的元素個數)+1
  // 如果剛好整除，會多配一個節點。
  size_type num_nodes = num_elements / buffer_size() + 1;

  // 一個 map 要管理幾個節點。最少 8 個，最多是 “所需節點數加 2”
  // （前後各預留一個，擴充時可用）。
  map_size = max(initial_map_size(), num_nodes + 2);
  map = map_allocator::allocate(map_size);
  // 以上配置出一個 “具有 map_size個節點” 的map。

  // 以下令nstart和nfinish指向map所擁有之全部節點的最中央區段。
  // 保持在最中央，可使頭尾兩端的擴充能量一樣大。每個節點可對應一個緩衝區。
  map_pointer nstart = map + (map_size - num_nodes) / 2;
  map_pointer nfinish = nstart + num_nodes - 1;

  map_pointer cur;
  __STL_TRY {
    // 為map內的每個現用節點配置緩衝區。所有緩衝區加起來就是deque的空間
   // （最後一個緩衝區可能留有一些餘裕）。
    for (cur = nstart; cur <= nfinish; ++cur)
      *cur = allocate_node();
  }
#    ifdef  __STL_USE_EXCEPTIONS
  catch(...) {
    // "commit or rollback" 語意：若非全部成功，就一個不留。
    for (map_pointer n = nstart; n < cur; ++n)
      deallocate_node(*n);
    map_allocator::deallocate(map, map_size);
    throw;
  }
#    endif /* __STL_USE_EXCEPTIONS */

  // 為deque內的兩個迭代器start和end 設定正確的內容。
  start.set_node(nstart);
  finish.set_node(nfinish);
  start.cur = start.first;      // first, cur都是public
  finish.cur = finish.first + num_elements % buffer_size();
  // 前面說過，如果剛好整除，會多配一個節點。
  // 此時即令cur指向這多配的一個節點（所對映之緩衝區）的起頭處。
}

// 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);  // 把deque的結構都產生並安排好
  map_pointer cur;
  __STL_TRY {
    // 為每個節點的緩衝區設定初值
    for (cur = start.node; cur < finish.node; ++cur)
      uninitialized_fill(*cur, *cur + buffer_size(), value);
    // 最後一個節點的設定稍有不同（因為尾端可能有備用空間，不必設初值）
    uninitialized_fill(finish.first, finish.cur, value);
  }
#     ifdef __STL_USE_EXCEPTIONS
  catch(...) {
    // "commit or rollback" 語意：若非全部成功，就一個不留。
    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 */

// 只有當 finish.cur == finish.last – 1 時才會被呼叫。
// 也就是說只有當最後一個緩衝區只剩一個備用元素空間時才會被呼叫。
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();        //  若符合某種條件則必須重換一個map
  *(finish.node + 1) = allocate_node();  // 配置一個新節點（緩衝區）
  __STL_TRY {
    construct(finish.cur, t_copy);        // 針對標的元素設值
    finish.set_node(finish.node + 1);     // 改變finish，令其指向新節點
    finish.cur = finish.first;            // 設定 finish 的狀態
  }
  __STL_UNWIND(deallocate_node(*(finish.node + 1)));
}

// 只有當start.cur == start.first時才會被呼叫。
// 也就是說只有當第一個緩衝區沒有任何備用元素時才會被呼叫。
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();       //  若符合某種條件則必須重換一個map
  *(start.node - 1) = allocate_node();    // 配置一個新節點（緩衝區）
  __STL_TRY {
    start.set_node(start.node - 1);       // 改變start，令其指向新節點
    start.cur = start.last - 1;           // 設定 start的狀態
    construct(start.cur, t_copy);         // 針對標的元素設值
  }
#    ifdef __STL_USE_EXCEPTIONS
  catch(...) {
    // "commit or rollback" 語意：若非全部成功，就一個不留。
    start.set_node(start.node + 1);
    start.cur = start.first;
    deallocate_node(*(start.node - 1));
    throw;
  }
#    endif /* __STL_USE_EXCEPTIONS */
}

// 只有當finish.cur == finish.first時才會被呼叫。
template <class T, class Alloc, size_t BufSize>
```

```
void deque<T, Alloc, BufSize>::pop_back_aux() {
  deallocate_node(finish.first);     // 釋放最後一個緩衝區
  finish.set_node(finish.node - 1); // 調整 finish 的狀態，使指向
  finish.cur = finish.last - 1;      //  上一個緩衝區的最後一個元素
  destroy(finish.cur);               // 將該元素解構。
}

// 只有當start.cur == start.last - 1時才會被呼叫。
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::pop_front_aux() {
  destroy(start.cur);                // 將第一緩衝區的第一個元素解構。
  deallocate_node(start.first);      // 釋放第一緩衝區。
  start.set_node(start.node + 1);    // 調整 start 的狀態，使指向
  start.cur = start.first;           //  下一個緩衝區的第一個元素。
}

#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;    // 安插點之前的元素個數
  value_type x_copy = x;
  if (index < size() / 2) {           // 如果安插點之前的元素個數比較少
    push_front(front());              // 在最前端加入與第一元素同值的元素。
    iterator front1 = start;          // 以下標示記號，然後進行元素搬移...
    ++front1;
    iterator front2 = front1;
    ++front2;
    pos = start + index;
    iterator pos1 = pos;
    ++pos1;
    copy(front2, pos1, front1);       // 元素搬移
  }
  else {                            // 安插點之後的元素個數比較少
    push_back(back());                // 在最尾端加入與最後元素同值的元素。
    iterator back1 = finish;          // 以下標示記號，然後進行元素搬移...
    --back1;
    iterator back2 = back1;
    --back2;
    pos = start + index;
    copy_backward(pos, back2, back1);     // 元素搬移
  }
  *pos = x_copy;  // 在安插點上設定新值
  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;
    // 配置一塊空間，準備給新map使用。
    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);
    // 把原map 內容拷貝過來。
    copy(start.node, finish.node + 1, new_nstart);
    // 釋放原map
    map_allocator::deallocate(map, map_size);
    // 設定新map的起始位址與大小
    map = new_map;
    map_size = new_map_size;
  }

  // 重新設定迭代器 start 和 finish
```

```
  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:
```