G++ 2.91.57，cygnus\cygwin-b20\include\g++**stl_bvector.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) 1996,1997
 * 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_BVECTOR_H
#define __SGI_STL_INTERNAL_BVECTOR_H

__STL_BEGIN_NAMESPACE

static const int __WORD_BIT = int(CHAR_BIT*sizeof(unsigned int));

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

struct __bit_reference {
  unsigned int* p;
  unsigned int mask;
  __bit_reference(unsigned int* x, unsigned int y) : p(x), mask(y) {}

public:
  __bit_reference() : p(0), mask(0) {}
```

```
  operator bool() const { return !(!(*p & mask)); }
  __bit_reference& operator=(bool x) {
    if (x)
      *p |= mask;
    else
      *p &= ~mask;
    return *this;
  }
  __bit_reference& operator=(const __bit_reference& x) { return *this = bool(x); }
  bool operator==(const __bit_reference& x) const {
    return bool(*this) == bool(x);
  }
  bool operator<(const __bit_reference& x) const {
    return bool(*this) < bool(x);
  }
  void flip() { *p ^= mask; }
};

inline void swap(__bit_reference x, __bit_reference y) {
  bool tmp = x;
  x = y;
  y = tmp;
}

struct __bit_iterator : public random_access_iterator<bool, ptrdiff_t>
{
  typedef __bit_reference  reference;
  typedef __bit_reference* pointer;
  typedef __bit_iterator iterator;

  unsigned int* p;
  unsigned int offset;
  void bump_up() {
    if (offset++ == __WORD_BIT - 1) {
      offset = 0;
      ++p;
    }
  }
  void bump_down() {
    if (offset-- == 0) {
      offset = __WORD_BIT - 1;
      --p;
    }
  }

  __bit_iterator() : p(0), offset(0) {}
  __bit_iterator(unsigned int* x, unsigned int y) : p(x), offset(y) {}
  reference operator*() const { return reference(p, 1U << offset); }
  iterator& operator++() {
```

```
    bump_up();
    return *this;
  }
  iterator operator++(int) {
    iterator tmp = *this;
    bump_up();
    return tmp;
  }
  iterator& operator--() {
    bump_down();
    return *this;
  }
  iterator operator--(int) {
    iterator tmp = *this;
    bump_down();
    return tmp;
  }
  iterator& operator+=(difference_type i) {
    difference_type n = i + offset;
    p += n / __WORD_BIT;
    n = n % __WORD_BIT;
    if (n < 0) {
      offset = (unsigned int) n + __WORD_BIT;
      --p;
    } else
      offset = (unsigned int) n;
    return *this;
  }
  iterator& operator-=(difference_type i) {
    *this += -i;
    return *this;
  }
  iterator operator+(difference_type i) const {
    iterator tmp = *this;
    return tmp += i;
  }
  iterator operator-(difference_type i) const {
    iterator tmp = *this;
    return tmp -= i;
  }
  difference_type operator-(iterator x) const {
    return __WORD_BIT * (p - x.p) + offset - x.offset;
  }
  reference operator[](difference_type i) { return *(*this + i); }
  bool operator==(const iterator& x) const {
    return p == x.p && offset == x.offset;
  }
  bool operator!=(const iterator& x) const {
    return p != x.p || offset != x.offset;
```

```
  }
  bool operator<(iterator x) const {
    return p < x.p || (p == x.p && offset < x.offset);
  }
};

struct __bit_const_iterator
  : public random_access_iterator<bool, ptrdiff_t>
{
  typedef bool             reference;
  typedef bool             const_reference;
  typedef const bool*       pointer;
  typedef __bit_const_iterator const_iterator;

  unsigned int* p;
  unsigned int offset;
  void bump_up() {
    if (offset++ == __WORD_BIT - 1) {
      offset = 0;
      ++p;
    }
  }
  void bump_down() {
    if (offset-- == 0) {
      offset = __WORD_BIT - 1;
      --p;
    }
  }

  __bit_const_iterator() : p(0), offset(0) {}
  __bit_const_iterator(unsigned int* x, unsigned int y) : p(x), offset(y) {}
  __bit_const_iterator(const __bit_iterator& x) : p(x.p), offset(x.offset) {}
  const_reference operator*() const {
    return __bit_reference(p, 1U << offset);
  }
  const_iterator& operator++() {
    bump_up();
    return *this;
  }
  const_iterator operator++(int) {
    const_iterator tmp = *this;
    bump_up();
    return tmp;
  }
  const_iterator& operator--() {
    bump_down();
    return *this;
  }
  const_iterator operator--(int) {
```

```
    const_iterator tmp = *this;
    bump_down();
    return tmp;
  }
  const_iterator& operator+=(difference_type i) {
    difference_type n = i + offset;
    p += n / __WORD_BIT;
    n = n % __WORD_BIT;
    if (n < 0) {
      offset = (unsigned int) n + __WORD_BIT;
      --p;
    } else
      offset = (unsigned int) n;
    return *this;
  }
  const_iterator& operator-=(difference_type i) {
    *this += -i;
    return *this;
  }
  const_iterator operator+(difference_type i) const {
    const_iterator tmp = *this;
    return tmp += i;
  }
  const_iterator operator-(difference_type i) const {
    const_iterator tmp = *this;
    return tmp -= i;
  }
  difference_type operator-(const_iterator x) const {
    return __WORD_BIT * (p - x.p) + offset - x.offset;
  }
  const_reference operator[](difference_type i) {
    return *(*this + i);
  }
  bool operator==(const const_iterator& x) const {
    return p == x.p && offset == x.offset;
  }
  bool operator!=(const const_iterator& x) const {
    return p != x.p || offset != x.offset;
  }
  bool operator<(const_iterator x) const {
    return p < x.p || (p == x.p && offset < x.offset);
  }
};

// 以下數行可能令你困惑。我們所要做的其實是宣告一個vector<T, Alloc>
// 的特化版本（partial specialization）— 如果我們擁有必要的編譯器支援的話。
// 否則我們就定義一個 class bit_vector，並令它使用default allocator。
//  不論哪一種情況，我們都以 typedef 適當定義了一個 "data_allocator"。
```

```
#if defined(__STL_CLASS_PARTIAL_SPECIALIZATION)
&& !defined(__STL_NEED_BOOL)
#define __SGI_STL_VECBOOL_TEMPLATE
#define __BVECTOR vector
#else
#undef __SGI_STL_VECBOOL_TEMPLATE
#define __BVECTOR bit_vector
#endif

#     ifdef __SGI_STL_VECBOOL_TEMPLATE
      __STL_END_NAMESPACE
#     include <stl_vector.h>
      __STL_BEGIN_NAMESPACE
template<class Alloc>
class vector<bool, Alloc>
#     else /* __SGI_STL_VECBOOL_TEMPLATE */
class bit_vector
#     endif /* __SGI_STL_VECBOOL_TEMPLATE */
{
#     ifdef __SGI_STL_VECBOOL_TEMPLATE
  typedef simple_alloc<unsigned int, Alloc> data_allocator;
#     else /* __SGI_STL_VECBOOL_TEMPLATE */
  typedef simple_alloc<unsigned int, alloc> data_allocator;
#     endif /* __SGI_STL_VECBOOL_TEMPLATE */
public:
  typedef bool value_type;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;
  typedef __bit_reference reference;
  typedef bool const_reference;
  typedef __bit_reference* pointer;
  typedef const bool* const_pointer;

  typedef __bit_iterator             iterator;
  typedef __bit_const_iterator        const_iterator;

#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:
  iterator start;
  iterator finish;
```

```
  unsigned int* end_of_storage;
  unsigned int* bit_alloc(size_type n) {
    return data_allocator::allocate((n + __WORD_BIT - 1)/__WORD_BIT);
  }
  void deallocate() {
    if (start.p)
      data_allocator::deallocate(start.p, end_of_storage - start.p);
  }
  void initialize(size_type n) {
    unsigned int* q = bit_alloc(n);
    end_of_storage = q + (n + __WORD_BIT - 1)/__WORD_BIT;
    start = iterator(q, 0);
    finish = start + difference_type(n);
  }
  void insert_aux(iterator position, bool x) {
    if (finish.p != end_of_storage) {
      copy_backward(position, finish, finish + 1);
      *position = x;
      ++finish;
    }
    else {
      size_type len = size() ? 2 * size() : __WORD_BIT;
      unsigned int* q = bit_alloc(len);
      iterator i = copy(begin(), position, iterator(q, 0));
      *i++ = x;
      finish = copy(position, end(), i);
      deallocate();
      end_of_storage = q + (len + __WORD_BIT - 1)/__WORD_BIT;
      start = iterator(q, 0);
    }
  }

#ifdef __STL_MEMBER_TEMPLATES
  template <class InputIterator>
  void initialize_range(InputIterator first, InputIterator last,
                        input_iterator_tag) {
    start = iterator();
    finish = iterator();
    end_of_storage = 0;
    for ( ; first != last; ++first)
      push_back(*first);
  }

  template <class ForwardIterator>
  void initialize_range(ForwardIterator first, ForwardIterator last,
                        forward_iterator_tag) {
    size_type n = 0;
    distance(first, last, n);
    initialize(n);
```

```
    copy(first, last, start);
  }

  template <class InputIterator>
  void insert_range(iterator pos,
                    InputIterator first, InputIterator last,
                    input_iterator_tag) {
    for ( ; first != last; ++first) {
      pos = insert(pos, *first);
      ++pos;
    }
  }

  template <class ForwardIterator>
  void insert_range(iterator position,
                    ForwardIterator first, ForwardIterator last,
                    forward_iterator_tag) {
    if (first != last) {
      size_type n = 0;
      distance(first, last, n);
      if (capacity() - size() >= n) {
        copy_backward(position, end(), finish + difference_type(n));
        copy(first, last, position);
        finish += difference_type(n);
      }
      else {
        size_type len = size() + max(size(), n);
        unsigned int* q = bit_alloc(len);
        iterator i = copy(begin(), position, iterator(q, 0));
        i = copy(first, last, i);
        finish = copy(position, end(), i);
        deallocate();
        end_of_storage = q + (len + __WORD_BIT - 1)/__WORD_BIT;
        start = iterator(q, 0);
      }
    }
  }

#endif /* __STL_MEMBER_TEMPLATES */

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

  reverse_iterator rbegin() { return reverse_iterator(end()); }
  const_reverse_iterator rbegin() const {
    return const_reverse_iterator(end());
```

```
  }
  reverse_iterator rend() { return reverse_iterator(begin()); }
  const_reverse_iterator rend() const {
    return const_reverse_iterator(begin());
  }

  size_type size() const { return size_type(end() - begin()); }
  size_type max_size() const { return size_type(-1); }
  size_type capacity() const {
    return size_type(const_iterator(end_of_storage, 0) - begin());
  }
  bool empty() const { return begin() == end(); }
  reference operator[](size_type n) {
    return *(begin() + difference_type(n));
  }
  const_reference operator[](size_type n) const {
    return *(begin() + difference_type(n));
  }
  __BVECTOR() : start(iterator()), finish(iterator()), end_of_storage(0) {}
  __BVECTOR(size_type n, bool value) {
    initialize(n);
    fill(start.p, end_of_storage, value ? ~0 : 0);
  }
  __BVECTOR(int n, bool value) {
    initialize(n);
    fill(start.p, end_of_storage, value ? ~0 : 0);
  }
  __BVECTOR(long n, bool value) {
    initialize(n);
    fill(start.p, end_of_storage, value ? ~0 : 0);
  }
  explicit __BVECTOR(size_type n) {
    initialize(n);
    fill(start.p, end_of_storage, 0);
  }
  __BVECTOR(const __BVECTOR& x) {
    initialize(x.size());
    copy(x.begin(), x.end(), start);
  }

#ifdef __STL_MEMBER_TEMPLATES
  template <class InputIterator>
  __BVECTOR(InputIterator first, InputIterator last) {
    initialize_range(first, last, iterator_category(first));
  }
#else /* __STL_MEMBER_TEMPLATES */
  __BVECTOR(const_iterator first, const_iterator last) {
    size_type n = 0;
    distance(first, last, n);
```

```
    initialize(n);
    copy(first, last, start);
  }
  __BVECTOR(const bool* first, const bool* last) {
    size_type n = 0;
    distance(first, last, n);
    initialize(n);
    copy(first, last, start);
  }
#endif /* __STL_MEMBER_TEMPLATES */

  ~__BVECTOR() { deallocate(); }
  __BVECTOR& operator=(const __BVECTOR& x) {
    if (&x == this) return *this;
    if (x.size() > capacity()) {
      deallocate();
      initialize(x.size());
    }
    copy(x.begin(), x.end(), begin());
    finish = begin() + difference_type(x.size());
    return *this;
  }
  void reserve(size_type n) {
    if (capacity() < n) {
      unsigned int* q = bit_alloc(n);
      finish = copy(begin(), end(), iterator(q, 0));
      deallocate();
      start = iterator(q, 0);
      end_of_storage = q + (n + __WORD_BIT - 1)/__WORD_BIT;
    }
  }
  reference front() { return *begin(); }
  const_reference front() const { return *begin(); }
  reference back() { return *(end() - 1); }
  const_reference back() const { return *(end() - 1); }
  void push_back(bool x) {
    if (finish.p != end_of_storage)
      *finish++ = x;
    else
      insert_aux(end(), x);
  }
  void swap(__BVECTOR& x) {
    __STD::swap(start, x.start);
    __STD::swap(finish, x.finish);
    __STD::swap(end_of_storage, x.end_of_storage);
  }
  iterator insert(iterator position, bool x = bool()) {
    difference_type n = position - begin();
    if (finish.p != end_of_storage && position == end())
```

```
      *finish++ = x;
    else
      insert_aux(position, x);
    return begin() + n;
  }

#ifdef __STL_MEMBER_TEMPLATES
  template <class InputIterator> void insert(iterator position,
                                             InputIterator first,
                                             InputIterator last) {
    insert_range(position, first, last, iterator_category(first));
  }
#else /* __STL_MEMBER_TEMPLATES */
  void insert(iterator position, const_iterator first,
              const_iterator last) {
    if (first == last) return;
    size_type n = 0;
    distance(first, last, n);
    if (capacity() - size() >= n) {
      copy_backward(position, end(), finish + n);
      copy(first, last, position);
      finish += n;
    }
    else {
      size_type len = size() + max(size(), n);
      unsigned int* q = bit_alloc(len);
      iterator i = copy(begin(), position, iterator(q, 0));
      i = copy(first, last, i);
      finish = copy(position, end(), i);
      deallocate();
      end_of_storage = q + (len + __WORD_BIT - 1)/__WORD_BIT;
      start = iterator(q, 0);
    }
  }

  void insert(iterator position, const bool* first, const bool* last) {
    if (first == last) return;
    size_type n = 0;
    distance(first, last, n);
    if (capacity() - size() >= n) {
      copy_backward(position, end(), finish + n);
      copy(first, last, position);
      finish += n;
    }
    else {
      size_type len = size() + max(size(), n);
      unsigned int* q = bit_alloc(len);
      iterator i = copy(begin(), position, iterator(q, 0));
      i = copy(first, last, i);
```

```
      finish = copy(position, end(), i);
      deallocate();
      end_of_storage = q + (len + __WORD_BIT - 1)/__WORD_BIT;
      start = iterator(q, 0);
    }
  }
#endif /* __STL_MEMBER_TEMPLATES */

  void insert(iterator position, size_type n, bool x) {
    if (n == 0) return;
    if (capacity() - size() >= n) {
      copy_backward(position, end(), finish + difference_type(n));
      fill(position, position + difference_type(n), x);
      finish += difference_type(n);
    }
    else {
      size_type len = size() + max(size(), n);
      unsigned int* q = bit_alloc(len);
      iterator i = copy(begin(), position, iterator(q, 0));
      fill_n(i, n, x);
      finish = copy(position, end(), i + difference_type(n));
      deallocate();
      end_of_storage = q + (len + __WORD_BIT - 1)/__WORD_BIT;
      start = iterator(q, 0);
    }
  }

  void insert(iterator pos, int n, bool x)  { insert(pos, (size_type)n, x); }
  void insert(iterator pos, long n, bool x) { insert(pos, (size_type)n, x); }

  void pop_back() { --finish; }
  iterator erase(iterator position) {
    if (position + 1 != end())
      copy(position + 1, end(), position);
    --finish;
    return position;
  }
  iterator erase(iterator first, iterator last) {
    finish = copy(last, end(), first);
    return first;
  }
  void resize(size_type new_size, bool x = bool()) {
    if (new_size < size())
      erase(begin() + difference_type(new_size), end());
    else
      insert(end(), new_size - size(), x);
  }
  void clear() { erase(begin(), end()); }
};
```

```
#ifdef __SGI_STL_VECBOOL_TEMPLATE

typedef vector<bool, alloc> bit_vector;

#else /* __SGI_STL_VECBOOL_TEMPLATE */

inline bool operator==(const bit_vector& x, const bit_vector& y) {
  return x.size() == y.size() && equal(x.begin(), x.end(), y.begin());
}

inline bool operator<(const bit_vector& x, const bit_vector& y) {
  return lexicographical_compare(x.begin(), x.end(), y.begin(), y.end());
}

#endif /* __SGI_STL_VECBOOL_TEMPLATE */

#undef __SGI_STL_VECBOOL_TEMPLATE
#undef __BVECTOR

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

__STL_END_NAMESPACE

#endif /* __SGI_STL_INTERNAL_BVECTOR_H */

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