G++ 2.91.57 · cygnus\cygwin-b20\include\g++**stl_vector.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
 * 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_VECTOR_H
#define __SGI_STL_INTERNAL_VECTOR_H

__STL_BEGIN_NAMESPACE

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

template <class T, class Alloc = alloc>  // jjhou: alloc defined in stl_alloc.h
class vector {
public:
  typedef T value_type;
  typedef value_type* pointer;
  typedef const value_type* const_pointer;
  typedef value_type* iterator;
  typedef const value_type* const_iterator;
```

```
  typedef value_type& reference;
  typedef const value_type& const_reference;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;

#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:
  typedef simple_alloc<value_type, Alloc> data_allocator;
  iterator start;
  iterator finish;
  iterator end_of_storage;
  void insert_aux(iterator position, const T& x);
  void deallocate() {
    if (start) data_allocator::deallocate(start, end_of_storage - start);
  }

  void fill_initialize(size_type n, const T& value) {
    start = allocate_and_fill(n, value);
    finish = start + n;
    end_of_storage = finish;
  }
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) / sizeof(T); }
  size_type capacity() const { return size_type(end_of_storage - begin()); }
  bool empty() const { return begin() == end(); }
  reference operator[](size_type n) { return *(begin() + n); }
  const_reference operator[](size_type n) const { return *(begin() + n); }
```

```
  vector() : start(0), finish(0), end_of_storage(0) {}
  vector(size_type n, const T& value) { fill_initialize(n, value); }
  vector(int n, const T& value) { fill_initialize(n, value); }
  vector(long n, const T& value) { fill_initialize(n, value); }
  explicit vector(size_type n) { fill_initialize(n, T()); }

  vector(const vector<T, Alloc>& x) {
    start = allocate_and_copy(x.end() - x.begin(), x.begin(), x.end());
    finish = start + (x.end() - x.begin());
    end_of_storage = finish;
  }
#ifdef __STL_MEMBER_TEMPLATES
  template <class InputIterator>
  vector(InputIterator first, InputIterator last) :
    start(0), finish(0), end_of_storage(0)
  {
    range_initialize(first, last, iterator_category(first));
  }
#else /* __STL_MEMBER_TEMPLATES */
  vector(const_iterator first, const_iterator last) {
    size_type n = 0;
    distance(first, last, n);
    start = allocate_and_copy(n, first, last);
    finish = start + n;
    end_of_storage = finish;
  }
#endif /* __STL_MEMBER_TEMPLATES */
  ~vector() {
    destroy(start, finish);  // destroy() defined where ?
    deallocate();  // jjhou:deallocate() is a member function of vector class
  }
  vector<T, Alloc>& operator=(const vector<T, Alloc>& x);
  void reserve(size_type n) {
    if (capacity() < n) {
      const size_type old_size = size();
      iterator tmp = allocate_and_copy(n, start, finish);
      destroy(start, finish);
      deallocate();
      start = tmp;
      finish = tmp + old_size;
      end_of_storage = start + n;
    }
  }
  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(const T& x) {
    if (finish != end_of_storage) {
```

```
      construct(finish, x);   // destroy() defined where ?
      ++finish;
    }
    else
      insert_aux(end(), x);
  }
  void swap(vector<T, Alloc>& x) {
    __STD::swap(start, x.start);
    __STD::swap(finish, x.finish);
    __STD::swap(end_of_storage, x.end_of_storage);
  }
  iterator insert(iterator position, const T& x) {
    size_type n = position - begin();
    if (finish != end_of_storage && position == end()) {
      construct(finish, x);
      ++finish;
    }
    else
      insert_aux(position, x);
    return begin() + n;
  }
  iterator insert(iterator position) { return insert(position, T()); }
#ifdef __STL_MEMBER_TEMPLATES
  template <class InputIterator>
  void insert(iterator position, InputIterator first, InputIterator last) {
    range_insert(position, first, last, iterator_category(first));
  }
#else /* __STL_MEMBER_TEMPLATES */
  void insert(iterator position,
              const_iterator first, const_iterator last);
#endif /* __STL_MEMBER_TEMPLATES */

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

  void pop_back() {
    --finish;
    destroy(finish);
  }
  iterator erase(iterator position) {
    if (position + 1 != end())
      copy(position + 1, finish, position);
    --finish;
    destroy(finish);
```

```
    return position;
  }
  iterator erase(iterator first, iterator last) {
    iterator i = copy(last, finish, first);
    destroy(i, finish);
    finish = finish - (last - first);
    return first;
  }
  void resize(size_type new_size, const T& x) {
    if (new_size < size())
      erase(begin() + new_size, end());
    else
      insert(end(), new_size - size(), x);
  }
  void resize(size_type new_size) { resize(new_size, T()); }
  void clear() { erase(begin(), end()); }

protected:
  iterator allocate_and_fill(size_type n, const T& x) {
    iterator result = data_allocator::allocate(n);
    __STL_TRY {
      uninitialized_fill_n(result, n, x);
      return result;
    }
    __STL_UNWIND(data_allocator::deallocate(result, n));
  }

#ifdef __STL_MEMBER_TEMPLATES
  template <class ForwardIterator>
  iterator allocate_and_copy(size_type n,
                             ForwardIterator first, ForwardIterator last) {
    iterator result = data_allocator::allocate(n);
    __STL_TRY {
      uninitialized_copy(first, last, result);
      return result;
    }
    __STL_UNWIND(data_allocator::deallocate(result, n));
  }
#else /* __STL_MEMBER_TEMPLATES */
  iterator allocate_and_copy(size_type n,
                             const_iterator first, const_iterator last) {
    iterator result = data_allocator::allocate(n);
    __STL_TRY {
      uninitialized_copy(first, last, result);
      return result;
    }
    __STL_UNWIND(data_allocator::deallocate(result, n));
  }
#endif /* __STL_MEMBER_TEMPLATES */
```

```
#ifdef __STL_MEMBER_TEMPLATES
  template <class InputIterator>
  void range_initialize(InputIterator first, InputIterator last,
                        input_iterator_tag) {
    for ( ; first != last; ++first)
      push_back(*first);
  }

  // This function is only called by the constructor.  We have to worry
  //  about resource leaks, but not about maintaining invariants.
  template <class ForwardIterator>
  void range_initialize(ForwardIterator first, ForwardIterator last,
                        forward_iterator_tag) {
    size_type n = 0;
    distance(first, last, n);
    start = allocate_and_copy(n, first, last);
    finish = start + n;
    end_of_storage = finish;
  }

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

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

#endif /* __STL_MEMBER_TEMPLATES */
};

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

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

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

template <class T, class Alloc>
inline void swap(vector<T, Alloc>& x, vector<T, Alloc>& y) {
```

```
  x.swap(y);
}

#endif /* __STL_FUNCTION_TMPL_PARTIAL_ORDER */

template <class T, class Alloc>
vector<T, Alloc>& vector<T, Alloc>::operator=(const vector<T, Alloc>& x) {
  if (&x != this) {
    if (x.size() > capacity()) {
      iterator tmp = allocate_and_copy(x.end() - x.begin(),
                                       x.begin(), x.end());
      destroy(start, finish);
      deallocate();
      start = tmp;
      end_of_storage = start + (x.end() - x.begin());
    }
    else if (size() >= x.size()) {
      iterator i = copy(x.begin(), x.end(), begin());
      destroy(i, finish);
    }
    else {
      copy(x.begin(), x.begin() + size(), start);
      uninitialized_copy(x.begin() + size(), x.end(), finish);
    }
    finish = start + x.size();
  }
  return *this;
}

template <class T, class Alloc>
void vector<T, Alloc>::insert_aux(iterator position, const T& x) {
  if (finish != end_of_storage) {
    construct(finish, *(finish - 1));
    ++finish;
    T x_copy = x;
    copy_backward(position, finish - 2, finish - 1);
    *position = x_copy;
  }
  else {
    const size_type old_size = size();
    const size_type len = old_size != 0 ? 2 * old_size : 1;
    iterator new_start = data_allocator::allocate(len);
    iterator new_finish = new_start;
    __STL_TRY {
      new_finish = uninitialized_copy(start, position, new_start);
      construct(new_finish, x);
      ++new_finish;
      new_finish = uninitialized_copy(position, finish, new_finish);
    }
```

```
#      ifdef  __STL_USE_EXCEPTIONS
    catch(...) {
      destroy(new_start, new_finish);
      data_allocator::deallocate(new_start, len);
      throw;
    }
#      endif /* __STL_USE_EXCEPTIONS */
    destroy(begin(), end());
    deallocate();
    start = new_start;
    finish = new_finish;
    end_of_storage = new_start + len;
  }
}

template <class T, class Alloc>
void vector<T, Alloc>::insert(iterator position, size_type n, const T& x) {
  if (n != 0) {
    if (size_type(end_of_storage - finish) >= n) {
      T x_copy = x;
      const size_type elems_after = finish - position;
      iterator old_finish = finish;
      if (elems_after > n) {
        uninitialized_copy(finish - n, finish, finish);
        finish += n;
        copy_backward(position, old_finish - n, old_finish);
        fill(position, position + n, x_copy);
      }
      else {
        uninitialized_fill_n(finish, n - elems_after, x_copy);
        finish += n - elems_after;
        uninitialized_copy(position, old_finish, finish);
        finish += elems_after;
        fill(position, old_finish, x_copy);
      }
    }
    else {
      const size_type old_size = size();
      const size_type len = old_size + max(old_size, n);
      iterator new_start = data_allocator::allocate(len);
      iterator new_finish = new_start;
      __STL_TRY {
        new_finish = uninitialized_copy(start, position, new_start);
        new_finish = uninitialized_fill_n(new_finish, n, x);
        new_finish = uninitialized_copy(position, finish, new_finish);
      }
#        ifdef  __STL_USE_EXCEPTIONS
      catch(...) {
```

```
      destroy(new_start, new_finish);
      data_allocator::deallocate(new_start, len);
      throw;
    }
#       endif /* __STL_USE_EXCEPTIONS */
    destroy(start, finish);
    deallocate();
    start = new_start;
    finish = new_finish;
    end_of_storage = new_start + len;
  }
 }
}

#ifdef __STL_MEMBER_TEMPLATES

template <class T, class Alloc> template <class InputIterator>
void vector<T, Alloc>::range_insert(iterator pos,
                                    InputIterator first, InputIterator last,
                                    input_iterator_tag) {
 for ( ; first != last; ++first) {
  pos = insert(pos, *first);
  ++pos;
 }
}

template <class T, class Alloc> template <class ForwardIterator>
void vector<T, Alloc>::range_insert(iterator position,
                                    ForwardIterator first,
                                    ForwardIterator last,
                                    forward_iterator_tag) {
 if (first != last) {
  size_type n = 0;
  distance(first, last, n);
  if (size_type(end_of_storage - finish) >= n) {
   const size_type elems_after = finish - position;
   iterator old_finish = finish;
   if (elems_after > n) {
    uninitialized_copy(finish - n, finish, finish);
    finish += n;
    copy_backward(position, old_finish - n, old_finish);
    copy(first, last, position);
   }
   else {
    ForwardIterator mid = first;
    advance(mid, elems_after);
    uninitialized_copy(mid, last, finish);
    finish += n - elems_after;
    uninitialized_copy(position, old_finish, finish);
```

```
      finish += elems_after;
      copy(first, mid, position);
    }
  }
  else {
    const size_type old_size = size();
    const size_type len = old_size + max(old_size, n);
    iterator new_start = data_allocator::allocate(len);
    iterator new_finish = new_start;
    __STL_TRY {
      new_finish = uninitialized_copy(start, position, new_start);
      new_finish = uninitialized_copy(first, last, new_finish);
      new_finish = uninitialized_copy(position, finish, new_finish);
    }
#       ifdef __STL_USE_EXCEPTIONS
    catch(...) {
      destroy(new_start, new_finish);
      data_allocator::deallocate(new_start, len);
      throw;
    }
#       endif /* __STL_USE_EXCEPTIONS */
    destroy(start, finish);
    deallocate();
    start = new_start;
    finish = new_finish;
    end_of_storage = new_start + len;
  }
 }
}

#else /* __STL_MEMBER_TEMPLATES */

template <class T, class Alloc>
void vector<T, Alloc>::insert(iterator position,
                          const_iterator first,
                          const_iterator last) {
 if (first != last) {
   size_type n = 0;
   distance(first, last, n);
   if (size_type(end_of_storage - finish) >= n) {
     const size_type elems_after = finish - position;
     iterator old_finish = finish;
     if (elems_after > n) {
       uninitialized_copy(finish - n, finish, finish);
       finish += n;
       copy_backward(position, old_finish - n, old_finish);
       copy(first, last, position);
     }
     else {
```

```
      uninitialized_copy(first + elems_after, last, finish);
      finish += n - elems_after;
      uninitialized_copy(position, old_finish, finish);
      finish += elems_after;
      copy(first, first + elems_after, position);
    }
   }
   else {
    const size_type old_size = size();
    const size_type len = old_size + max(old_size, n);
    iterator new_start = data_allocator::allocate(len);
    iterator new_finish = new_start;
    __STL_TRY {
      new_finish = uninitialized_copy(start, position, new_start);
      new_finish = uninitialized_copy(first, last, new_finish);
      new_finish = uninitialized_copy(position, finish, new_finish);
    }
#       ifdef __STL_USE_EXCEPTIONS
    catch(...) {
      destroy(new_start, new_finish);
      data_allocator::deallocate(new_start, len);
      throw;
    }
#       endif /* __STL_USE_EXCEPTIONS */
    destroy(start, finish);
    deallocate();
    start = new_start;
    finish = new_finish;
    end_of_storage = new_start + len;
   }
  }
}

#endif /* __STL_MEMBER_TEMPLATES */

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

__STL_END_NAMESPACE

#endif /* __SGI_STL_INTERNAL_VECTOR_H */

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