G++ 2.91.57，cygnus\cygwin-b20\include\g++**stl_multimap.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_MULTIMAP_H
#define __SGI_STL_INTERNAL_MULTIMAP_H

__STL_BEGIN_NAMESPACE

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

#ifndef __STL_LIMITED_DEFAULT_TEMPLATES
template <class Key, class T, class Compare = less<Key>, class Alloc = alloc>
#else
template <class Key, class T, class Compare, class Alloc = alloc>
#endif
class multimap {
public:

// typedefs:
```

```
  typedef Key key_type;
  typedef T data_type;
  typedef T mapped_type;
  typedef pair<const Key, T> value_type;
  typedef Compare key_compare;

  class value_compare : public binary_function<value_type, value_type, bool>
{
  friend class multimap<Key, T, Compare, Alloc>;
  protected:
    Compare comp;
    value_compare(Compare c) : comp(c) {}
  public:
    bool operator()(const value_type& x, const value_type& y) const {
      return comp(x.first, y.first);
    }
  };

private:
  typedef rb_tree<key_type, value_type,
                  select1st<value_type>, key_compare, Alloc> rep_type;
  rep_type t;  // red-black tree representing multimap
public:
  typedef typename rep_type::pointer pointer;
  typedef typename rep_type::const_pointer const_pointer;
  typedef typename rep_type::reference reference;
  typedef typename rep_type::const_reference const_reference;
  typedef typename rep_type::iterator iterator;
  typedef typename rep_type::const_iterator const_iterator;
  typedef typename rep_type::reverse_iterator reverse_iterator;
  typedef typename rep_type::const_reverse_iterator const_reverse_iterator;
  typedef typename rep_type::size_type size_type;
  typedef typename rep_type::difference_type difference_type;

// allocation/deallocation

  multimap() : t(Compare()) { }
  explicit multimap(const Compare& comp) : t(comp) { }

#ifdef __STL_MEMBER_TEMPLATES
  template <class InputIterator>
  multimap(InputIterator first, InputIterator last)
    : t(Compare()) { t.insert_equal(first, last); }

  template <class InputIterator>
  multimap(InputIterator first, InputIterator last, const Compare& comp)
    : t(comp) { t.insert_equal(first, last); }
#else
```

```
  multimap(const value_type* first, const value_type* last)
    : t(Compare()) { t.insert_equal(first, last); }
  multimap(const value_type* first, const value_type* last,
           const Compare& comp)
    : t(comp) { t.insert_equal(first, last); }

  multimap(const_iterator first, const_iterator last)
    : t(Compare()) { t.insert_equal(first, last); }
  multimap(const_iterator first, const_iterator last, const Compare& comp)
    : t(comp) { t.insert_equal(first, last); }
#endif /* __STL_MEMBER_TEMPLATES */

  multimap(const multimap<Key, T, Compare, Alloc>& x) : t(x.t) { }
  multimap<Key, T, Compare, Alloc>&
  operator=(const multimap<Key, T, Compare, Alloc>& x) {
    t = x.t;
    return *this;
  }

  // accessors:

  key_compare key_comp() const { return t.key_comp(); }
  value_compare value_comp() const { return value_compare(t.key_comp()); }
  iterator begin() { return t.begin(); }
  const_iterator begin() const { return t.begin(); }
  iterator end() { return t.end(); }
  const_iterator end() const { return t.end(); }
  reverse_iterator rbegin() { return t.rbegin(); }
  const_reverse_iterator rbegin() const { return t.rbegin(); }
  reverse_iterator rend() { return t.rend(); }
  const_reverse_iterator rend() const { return t.rend(); }
  bool empty() const { return t.empty(); }
  size_type size() const { return t.size(); }
  size_type max_size() const { return t.max_size(); }
  void swap(multimap<Key, T, Compare, Alloc>& x) { t.swap(x.t); }

  // insert/erase

  iterator insert(const value_type& x) { return t.insert_equal(x); }
  iterator insert(iterator position, const value_type& x) {
    return t.insert_equal(position, x);
  }
#ifdef __STL_MEMBER_TEMPLATES
  template <class InputIterator>
  void insert(InputIterator first, InputIterator last) {
    t.insert_equal(first, last);
  }
#else
  void insert(const value_type* first, const value_type* last) {
```

```
    t.insert_equal(first, last);
  }
  void insert(const_iterator first, const_iterator last) {
    t.insert_equal(first, last);
  }
#endif /* __STL_MEMBER_TEMPLATES */
  void erase(iterator position) { t.erase(position); }
  size_type erase(const key_type& x) { return t.erase(x); }
  void erase(iterator first, iterator last) { t.erase(first, last); }
  void clear() { t.clear(); }

  // multimap operations:

  iterator find(const key_type& x) { return t.find(x); }
  const_iterator find(const key_type& x) const { return t.find(x); }
  size_type count(const key_type& x) const { return t.count(x); }
  iterator lower_bound(const key_type& x) {return t.lower_bound(x); }
  const_iterator lower_bound(const key_type& x) const {
    return t.lower_bound(x);
  }
  iterator upper_bound(const key_type& x) {return t.upper_bound(x); }
  const_iterator upper_bound(const key_type& x) const {
    return t.upper_bound(x);
  }
   pair<iterator,iterator> equal_range(const key_type& x) {
    return t.equal_range(x);
  }
  pair<const_iterator,const_iterator> equal_range(const key_type& x) const {
    return t.equal_range(x);
  }
  friend bool operator== __STL_NULL_TMPL_ARGS (const multimap&,
                                               const multimap&);
  friend bool operator< __STL_NULL_TMPL_ARGS (const multimap&,
                                              const multimap&);
};

template <class Key, class T, class Compare, class Alloc>
inline bool operator==(const multimap<Key, T, Compare, Alloc>& x,
                       const multimap<Key, T, Compare, Alloc>& y) {
  return x.t == y.t;
}

template <class Key, class T, class Compare, class Alloc>
inline bool operator<(const multimap<Key, T, Compare, Alloc>& x,
                      const multimap<Key, T, Compare, Alloc>& y) {
  return x.t < y.t;
}

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER
```

```
template <class Key, class T, class Compare, class Alloc>
inline void swap(multimap<Key, T, Compare, Alloc>& x,
                 multimap<Key, T, Compare, Alloc>& y) {
  x.swap(y);
}

#endif /* __STL_FUNCTION_TMPL_PARTIAL_ORDER */

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

__STL_END_NAMESPACE

#endif /* __SGI_STL_INTERNAL_MULTIMAP_H */

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