G++ 2.91.57，cygnus\cygwin-b20\include\g++**stl_multiset.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_MULTISET_H
#define __SGI_STL_INTERNAL_MULTISET_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 Compare = less<Key>, class Alloc = alloc>
#else
template <class Key, class Compare, class Alloc = alloc>
#endif
class multiset {
public:
  // typedefs:
```

```
  typedef Key key_type;
  typedef Key value_type;
  typedef Compare key_compare;
  typedef Compare value_compare;
private:
  typedef rb_tree<key_type, value_type,
                  identity<value_type>, key_compare, Alloc> rep_type;
  rep_type t;  // red-black tree representing multiset
public:
  typedef typename rep_type::const_pointer pointer;
  typedef typename rep_type::const_pointer const_pointer;
  typedef typename rep_type::const_reference reference;
  typedef typename rep_type::const_reference const_reference;
  typedef typename rep_type::const_iterator iterator;
  typedef typename rep_type::const_iterator const_iterator;
  typedef typename rep_type::const_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
  // 注意， multiset 一定使用 insert_equal() 而不使用 insert_unique()。
  // set 才使用 insert_unique()。
  multiset() : t(Compare()) {}
  explicit multiset(const Compare& comp) : t(comp) {}

#ifdef __STL_MEMBER_TEMPLATES
  template <class InputIterator>
  multiset(InputIterator first, InputIterator last)
    : t(Compare()) { t.insert_equal(first, last); }
  template <class InputIterator>
  multiset(InputIterator first, InputIterator last, const Compare& comp)
    : t(comp) { t.insert_equal(first, last); }
#else
  multiset(const value_type* first, const value_type* last)
    : t(Compare()) { t.insert_equal(first, last); }
  multiset(const value_type* first, const value_type* last,
           const Compare& comp)
    : t(comp) { t.insert_equal(first, last); }

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

  multiset(const multiset<Key, Compare, Alloc>& x) : t(x.t) {}
  multiset<Key, Compare, Alloc>&
  operator=(const multiset<Key, Compare, Alloc>& x) {
```

```
    t = x.t;
    return *this;
  }

  // accessors:

  key_compare key_comp() const { return t.key_comp(); }
  value_compare value_comp() const { return t.key_comp(); }
  iterator begin() const { return t.begin(); }
  iterator end() const { return t.end(); }
  reverse_iterator rbegin() const { return t.rbegin(); }
  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(multiset<Key, 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) {
    typedef typename rep_type::iterator rep_iterator;
    return t.insert_equal((rep_iterator&)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) {
    typedef typename rep_type::iterator rep_iterator;
    t.erase((rep_iterator&)position);
  }
  size_type erase(const key_type& x) {
    return t.erase(x);
  }
  void erase(iterator first, iterator last) {
    typedef typename rep_type::iterator rep_iterator;
    t.erase((rep_iterator&)first, (rep_iterator&)last);
```

```
  }
  void clear() { t.clear(); }

  // multiset operations:

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

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

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

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

template <class Key, class Compare, class Alloc>
inline void swap(multiset<Key, Compare, Alloc>& x,
                 multiset<Key, 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_MULTISET_H */

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