```
G++ 2.91.57，cygnus\cygwin-b20\include\g++\stl_hash_map.h 完整列表
/*
 * 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.
 *
 *
 * 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.
 *
 */

/* NOTE: This is an internal header file, included by other STL headers.
 *   You should not attempt to use it directly.
 */

#ifndef __SGI_STL_INTERNAL_HASH_MAP_H
#define __SGI_STL_INTERNAL_HASH_MAP_H


__STL_BEGIN_NAMESPACE

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

#ifndef __STL_LIMITED_DEFAULT_TEMPLATES
// hash<> 是個 function object，定義於 <stl_hash_fun.h> 中
// 例：hash<int>::operator()(int x) const { return x; }
template <class Key, class T, class HashFcn = hash<Key>,
        class EqualKey = equal_to<Key>,
        class Alloc = alloc>
#else
template <class Key, class T, class HashFcn, class EqualKey,
         class Alloc = alloc>
#endif
class hash_map
{
private:
  // 以下使用的 select1st<> 定義於 <stl_function.h> 中。
  typedef hashtable<pair<const Key, T>, Key, HashFcn,
                    select1st<pair<const Key, T> >, EqualKey, Alloc> ht;
  ht rep;    // 底層機制以 hash table 完成

public:
  typedef typename ht::key_type key_type;
  typedef T data_type;
  typedef T mapped_type;
  typedef typename ht::value_type value_type;
  typedef typename ht::hasher hasher;
  typedef typename ht::key_equal key_equal;

  typedef typename ht::size_type size_type;
  typedef typename ht::difference_type difference_type;
  typedef typename ht::pointer pointer;
  typedef typename ht::const_pointer const_pointer;
  typedef typename ht::reference reference;
  typedef typename ht::const_reference const_reference;

  typedef typename ht::iterator iterator;
  typedef typename ht::const_iterator const_iterator;

  hasher hash_funct() const { return rep.hash_funct(); }
  key_equal key_eq() const { return rep.key_eq(); }

public:
  // 預設使用大小為 100 的表格。將由hash table 調整為最接近且較大之質數
  hash_map() : rep(100, hasher(), key_equal()) {}
  explicit hash_map(size_type n) : rep(n, hasher(), key_equal()) {}
  hash_map(size_type n, const hasher& hf) : rep(n, hf, key_equal()) {}
  hash_map(size_type n, const hasher& hf, const key_equal& eql)
    : rep(n, hf, eql) {}

#ifdef __STL_MEMBER_TEMPLATES
  // 以下，安插動作全部使用 insert_unique()，不允許鍵值重複。
  template <class InputIterator>
  hash_map(InputIterator f, InputIterator l)
    : rep(100, hasher(), key_equal()) { rep.insert_unique(f, l); }
  template <class InputIterator>
  hash_map(InputIterator f, InputIterator l, size_type n)
    : rep(n, hasher(), key_equal()) { rep.insert_unique(f, l); }
  template <class InputIterator>
  hash_map(InputIterator f, InputIterator l, size_type n,
          const hasher& hf)
    : rep(n, hf, key_equal()) { rep.insert_unique(f, l); }
  template <class InputIterator>
  hash_map(InputIterator f, InputIterator l, size_type n,
          const hasher& hf, const key_equal& eql)
    : rep(n, hf, eql) { rep.insert_unique(f, l); }

#else
  hash_map(const value_type* f, const value_type* l)
    : rep(100, hasher(), key_equal()) { rep.insert_unique(f, l); }
  hash_map(const value_type* f, const value_type* l, size_type n)
    : rep(n, hasher(), key_equal()) { rep.insert_unique(f, l); }
  hash_map(const value_type* f, const value_type* l, size_type n,
          const hasher& hf)
    : rep(n, hf, key_equal()) { rep.insert_unique(f, l); }
  hash_map(const value_type* f, const value_type* l, size_type n,
          const hasher& hf, const key_equal& eql)
    : rep(n, hf, eql) { rep.insert_unique(f, l); }

  hash_map(const_iterator f, const_iterator l)
    : rep(100, hasher(), key_equal()) { rep.insert_unique(f, l); }
  hash_map(const_iterator f, const_iterator l, size_type n)
    : rep(n, hasher(), key_equal()) { rep.insert_unique(f, l); }
  hash_map(const_iterator f, const_iterator l, size_type n,
          const hasher& hf)
    : rep(n, hf, key_equal()) { rep.insert_unique(f, l); }
  hash_map(const_iterator f, const_iterator l, size_type n,
          const hasher& hf, const key_equal& eql)
    : rep(n, hf, eql) { rep.insert_unique(f, l); }
#endif /*__STL_MEMBER_TEMPLATES */

public:
  // 所有操作幾乎都有 hash table 對應版本。轉呼叫就行。
  size_type size() const { return rep.size(); }
  size_type max_size() const { return rep.max_size(); }
  bool empty() const { return rep.empty(); }
  void swap(hash_map& hs) { rep.swap(hs.rep); }
  friend bool
  operator== __STL_NULL_TMPL_ARGS (const hash_map&, const hash_map&);

  iterator begin() { return rep.begin(); }
  iterator end() { return rep.end(); }
  const_iterator begin() const { return rep.begin(); }
  const_iterator end() const { return rep.end(); }

public:
  pair<iterator, bool> insert(const value_type& obj)
    { return rep.insert_unique(obj); }
#ifdef __STL_MEMBER_TEMPLATES
```

```
  template <class InputIterator>
  void insert(InputIterator f, InputIterator l) { rep.insert_unique(f,l); }
#else
  void insert(const value_type* f, const value_type* l) {
    rep.insert_unique(f,l);
  }
  void insert(const_iterator f, const_iterator l) { rep.insert_unique(f, l); }
#endif /*__STL_MEMBER_TEMPLATES */
  pair<iterator, bool> insert_noresize(const value_type& obj)
    { return rep.insert_unique_noresize(obj); }

  iterator find(const key_type& key) { return rep.find(key); }
  const_iterator find(const key_type& key) const { return rep.find(key); }

  T& operator[](const key_type& key) {
    return rep.find_or_insert(value_type(key, T())).second;
  }

  size_type count(const key_type& key) const { return rep.count(key); }

  pair<iterator, iterator> equal_range(const key_type& key)
    { return rep.equal_range(key); }
  pair<const_iterator, const_iterator> equal_range(const key_type& key) const
    { return rep.equal_range(key); }

  size_type erase(const key_type& key) {return rep.erase(key); }
  void erase(iterator it) { rep.erase(it); }
  void erase(iterator f, iterator l) { rep.erase(f, l); }
  void clear() { rep.clear(); }

public:
  void resize(size_type hint) { rep.resize(hint); }
  size_type bucket_count() const { return rep.bucket_count(); }
  size_type max_bucket_count() const { return rep.max_bucket_count(); }
  size_type elems_in_bucket(size_type n) const
    { return rep.elems_in_bucket(n); }
};

template <class Key, class T, class HashFcn, class EqualKey, class Alloc>
inline bool operator==(const hash_map<Key, T, HashFcn, EqualKey, Alloc>& hm1,
                       const hash_map<Key, T, HashFcn, EqualKey, Alloc>& hm2)
{
  return hm1.rep == hm2.rep;
}

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

template <class Key, class T, class HashFcn, class EqualKey, class Alloc>
inline void swap(hash_map<Key, T, HashFcn, EqualKey, Alloc>& hm1,
                 hash_map<Key, T, HashFcn, EqualKey, Alloc>& hm2)
{
  hm1.swap(hm2);
}

#endif /* __STL_FUNCTION_TMPL_PARTIAL_ORDER */

#ifndef __STL_LIMITED_DEFAULT_TEMPLATES
template <class Key, class T, class HashFcn = hash<Key>,
          class EqualKey = equal_to<Key>,
          class Alloc = alloc>
#else
template <class Key, class T, class HashFcn, class EqualKey,
          class Alloc = alloc>
#endif
class hash_multimap
{
private:
  typedef hashtable<pair<const Key, T>, Key, HashFcn,
                    select1st<pair<const Key, T> >, EqualKey, Alloc> ht;
  ht rep;

public:
  typedef typename ht::key_type key_type;
  typedef T data_type;
  typedef T mapped_type;
  typedef typename ht::value_type value_type;
  typedef typename ht::hasher hasher;
  typedef typename ht::key_equal key_equal;

  typedef typename ht::size_type size_type;
  typedef typename ht::difference_type difference_type;
  typedef typename ht::pointer pointer;
  typedef typename ht::const_pointer const_pointer;
  typedef typename ht::reference reference;
  typedef typename ht::const_reference const_reference;

  typedef typename ht::iterator iterator;
  typedef typename ht::const_iterator const_iterator;

  hasher hash_funct() const { return rep.hash_funct(); }
  key_equal key_eq() const { return rep.key_eq(); }

public:
  // 預設使用大小為 100 的表格。將被hash table 調整為最接近且較大之質數
  hash_multimap() : rep(100, hasher(), key_equal()) {}
  explicit hash_multimap(size_type n) : rep(n, hasher(), key_equal()) {}
  hash_multimap(size_type n, const hasher& hf) : rep(n, hf, key_equal()) {}
  hash_multimap(size_type n, const hasher& hf, const key_equal& eql)
```

```
    : rep(n, hf, eql) {}

#ifdef __STL_MEMBER_TEMPLATES
// 以下，安插動作全部使用 insert_equal()，允許鍵值重複。
  template <class InputIterator>
  hash_multimap(InputIterator f, InputIterator l)
    : rep(100, hasher(), key_equal()) { rep.insert_equal(f, l); }
  template <class InputIterator>
  hash_multimap(InputIterator f, InputIterator l, size_type n)
    : rep(n, hasher(), key_equal()) { rep.insert_equal(f, l); }
  template <class InputIterator>
  hash_multimap(InputIterator f, InputIterator l, size_type n,
                const hasher& hf)
    : rep(n, hf, key_equal()) { rep.insert_equal(f, l); }
  template <class InputIterator>
  hash_multimap(InputIterator f, InputIterator l, size_type n,
                const hasher& hf, const key_equal& eql)
    : rep(n, hf, eql) { rep.insert_equal(f, l); }

#else
  hash_multimap(const value_type* f, const value_type* l)
    : rep(100, hasher(), key_equal()) { rep.insert_equal(f, l); }
  hash_multimap(const value_type* f, const value_type* l, size_type n)
    : rep(n, hasher(), key_equal()) { rep.insert_equal(f, l); }
  hash_multimap(const value_type* f, const value_type* l, size_type n,
                const hasher& hf)
    : rep(n, hf, key_equal()) { rep.insert_equal(f, l); }
  hash_multimap(const value_type* f, const value_type* l, size_type n,
                const hasher& hf, const key_equal& eql)
    : rep(n, hf, eql) { rep.insert_equal(f, l); }

  hash_multimap(const_iterator f, const_iterator l)
    : rep(100, hasher(), key_equal()) { rep.insert_equal(f, l); }
  hash_multimap(const_iterator f, const_iterator l, size_type n)
    : rep(n, hasher(), key_equal()) { rep.insert_equal(f, l); }
  hash_multimap(const_iterator f, const_iterator l, size_type n,
                const hasher& hf)
    : rep(n, hf, key_equal()) { rep.insert_equal(f, l); }
  hash_multimap(const_iterator f, const_iterator l, size_type n,
                const hasher& hf, const key_equal& eql)
    : rep(n, hf, eql) { rep.insert_equal(f, l); }
#endif /*__STL_MEMBER_TEMPLATES */

public:
  // 所有操作幾乎都有 hash table 的對應版本，轉呼叫就好。
  size_type size() const { return rep.size(); }
  size_type max_size() const { return rep.max_size(); }
  bool empty() const { return rep.empty(); }
  void swap(hash_multimap& hs) { rep.swap(hs.rep); }
  friend bool
  operator== __STL_NULL_TMPL_ARGS (const hash_multimap&, const hash_multimap&);

  iterator begin() { return rep.begin(); }
  iterator end() { return rep.end(); }
  const_iterator begin() const { return rep.begin(); }
  const_iterator end() const { return rep.end(); }

public:
  iterator insert(const value_type& obj) { return rep.insert_equal(obj); }
#ifdef __STL_MEMBER_TEMPLATES
  template <class InputIterator>
  void insert(InputIterator f, InputIterator l) { rep.insert_equal(f,l); }
#else
  void insert(const value_type* f, const value_type* l) {
    rep.insert_equal(f,l);
  }
  void insert(const_iterator f, const_iterator l) { rep.insert_equal(f, l); }
#endif /*__STL_MEMBER_TEMPLATES */
  iterator insert_noresize(const value_type& obj)
    { return rep.insert_equal_noresize(obj); }

  iterator find(const key_type& key) { return rep.find(key); }
  const_iterator find(const key_type& key) const { return rep.find(key); }

  size_type count(const key_type& key) const { return rep.count(key); }

  pair<iterator, iterator> equal_range(const key_type& key)
    { return rep.equal_range(key); }
  pair<const_iterator, const_iterator> equal_range(const key_type& key) const
    { return rep.equal_range(key); }

  size_type erase(const key_type& key) {return rep.erase(key); }
  void erase(iterator it) { rep.erase(it); }
  void erase(iterator f, iterator l) { rep.erase(f, l); }
  void clear() { rep.clear(); }

public:
  void resize(size_type hint) { rep.resize(hint); }
  size_type bucket_count() const { return rep.bucket_count(); }
  size_type max_bucket_count() const { return rep.max_bucket_count(); }
  size_type elems_in_bucket(size_type n) const
    { return rep.elems_in_bucket(n); }
};

template <class Key, class T, class HF, class EqKey, class Alloc>
inline bool operator==(const hash_multimap<Key, T, HF, EqKey, Alloc>& hm1,
                       const hash_multimap<Key, T, HF, EqKey, Alloc>& hm2)
{
  return hm1.rep == hm2.rep;
}

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

template <class Key, class T, class HashFcn, class EqualKey, class Alloc>
inline void swap(hash_multimap<Key, T, HashFcn, EqualKey, Alloc>& hm1,
              hash_multimap<Key, T, HashFcn, EqualKey, Alloc>& hm2)
{
  hm1.swap(hm2);
}

#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_HASH_MAP_H */

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