G++ 2.91.57，cygnus\cygwin-b20\include\g++\stl_hashtable.h 完整列表

```
/*
 * 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.
 *
 *
 * 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_HASHTABLE_H
#define __SGI_STL_INTERNAL_HASHTABLE_H

// Hashtable class 用來實作 hashed associative containers
// hash_set, hash_map, hash_multiset, 和 hash_multimap.

#include <stl_algobase.h>
#include <stl_alloc.h>
#include <stl_construct.h>
#include <stl_tempbuf.h>
#include <stl_algo.h>
#include <stl_uninitialized.h>
#include <stl_function.h>
#include <stl_vector.h>
#include <stl_hash_fun.h>

__STL_BEGIN_NAMESPACE
```

```
template <class Value>
struct __hashtable_node
{
  // 既以vector來實作hash table，何必需要next 指標？
  // 這是因為SGI 的實作方式賦予每個bucket 以一個對應串列，所以每個bucket
  // 可能代表一系列節點，而不只是一個節點。
  // 這是所謂的 separate chaining 技巧。
  __hashtable_node* next;
  Value val;
};

template <class Value, class Key, class HashFcn,
        class ExtractKey, class EqualKey, class Alloc = alloc>
class hashtable;

// 由於 __hashtable_iterator 和 __hashtable_const_iterator 兩者會
// 互相使用，因此必須在下面先做宣告，否則難以編譯。
template <class Value, class Key, class HashFcn,
        class ExtractKey, class EqualKey, class Alloc>
struct __hashtable_iterator;

template <class Value, class Key, class HashFcn,
        class ExtractKey, class EqualKey, class Alloc>
struct __hashtable_const_iterator;

template <class Value, class Key, class HashFcn,
        class ExtractKey, class EqualKey, class Alloc>
struct __hashtable_iterator {
  typedef hashtable<Value, Key, HashFcn, ExtractKey, EqualKey, Alloc>
        hashtable;
  typedef __hashtable_iterator<Value, Key, HashFcn,
                          ExtractKey, EqualKey, Alloc>
        iterator;
  typedef __hashtable_const_iterator<Value, Key, HashFcn,
                               ExtractKey, EqualKey, Alloc>
        const_iterator;
  typedef __hashtable_node<Value> node;

  typedef forward_iterator_tag iterator_category;
  typedef Value value_type;
  typedef ptrdiff_t difference_type;
  typedef size_t size_type;
  typedef Value& reference;
  typedef Value* pointer;

  node* cur;      // 迭代器目前所指之節點
  hashtable* ht;  // 保持對容器的連結關係（因為可能需要從bucket 跳到bucket）

  __hashtable_iterator(node* n, hashtable* tab) : cur(n), ht(tab) {}
  __hashtable_iterator() {}
  reference operator*() const { return cur->val; }
#ifndef __SGI_STL_NO_ARROW_OPERATOR
  pointer operator->() const { return &(operator*()); }
#endif /* __SGI_STL_NO_ARROW_OPERATOR */
  iterator& operator++();
  iterator operator++(int);
  bool operator==(const iterator& it) const { return cur == it.cur; }
  bool operator!=(const iterator& it) const { return cur != it.cur; }
};


template <class Value, class Key, class HashFcn,
        class ExtractKey, class EqualKey, class Alloc>
struct __hashtable_const_iterator {
  typedef hashtable<Value, Key, HashFcn, ExtractKey, EqualKey, Alloc>
        hashtable;
  typedef __hashtable_iterator<Value, Key, HashFcn,
                              ExtractKey, EqualKey, Alloc>
        iterator;
  typedef __hashtable_const_iterator<Value, Key, HashFcn,
                                   ExtractKey, EqualKey, Alloc>
        const_iterator;
  typedef __hashtable_node<Value> node;

  typedef forward_iterator_tag iterator_category;  // 注意
  typedef Value value_type;
  typedef ptrdiff_t difference_type;
  typedef size_t size_type;
  typedef const Value& reference;
  typedef const Value* pointer;

  const node* cur;
  const hashtable* ht;

  __hashtable_const_iterator(const node* n, const hashtable* tab)
    : cur(n), ht(tab) {}
  __hashtable_const_iterator() {}
  __hashtable_const_iterator(const iterator& it) : cur(it.cur), ht(it.ht) {}
  reference operator*() const { return cur->val; }
#ifndef __SGI_STL_NO_ARROW_OPERATOR
  pointer operator->() const { return &(operator*()); }
#endif /* __SGI_STL_NO_ARROW_OPERATOR */
  const_iterator& operator++();
  const_iterator operator++(int);
  bool operator==(const const_iterator& it) const { return cur == it.cur; }
  bool operator!=(const const_iterator& it) const { return cur != it.cur; }
};
```

```
// 注意：假設 long 至少有 32 bits。
static const int __stl_num_primes = 28;
static const unsigned long __stl_prime_list[__stl_num_primes] =
{
  53,         97,           193,         389,       769,
  1543,       3079,         6151,        12289,     24593,
  49157,      98317,        196613,      393241,    786433,
  1572869,    3145739,      6291469,     12582917,  25165843,
  50331653,   100663319,    201326611,   402653189, 805306457,
  1610612741, 3221225473ul, 4294967291ul
};

// 以下找出上述 28 個質數之中，最接近並大於n的那個質數
inline unsigned long __stl_next_prime(unsigned long n)
{
  const unsigned long* first = __stl_prime_list;
  const unsigned long* last = __stl_prime_list + __stl_num_primes;
  const unsigned long* pos = lower_bound(first, last, n);
  // 以上，lower_bound() 是泛型演算法
  // 使用 lower_bound()，序列需先排序。沒問題，上述陣列已排序。
  return pos == last ? *(last - 1) : *pos;
}


template <class Value, class Key, class HashFcn,
          class ExtractKey, class EqualKey,
          class Alloc>  // 先前宣告時，已給予 Alloc 以預設值 alloc
class hashtable {
public:
  // 為 template 型別參數重新定義一個名稱（何必！）
  typedef Key key_type;
  typedef Value value_type;
  typedef HashFcn hasher;
  typedef EqualKey key_equal;

  typedef size_t            size_type;
  typedef ptrdiff_t         difference_type;
  typedef value_type*       pointer;
  typedef const value_type* const_pointer;
  typedef value_type&       reference;
  typedef const value_type& const_reference;

  hasher hash_funct() const { return hash; }
  key_equal key_eq() const { return equals; }

private:
  // 以下三者都是function objects。<stl_hash_fun.h> 中定義有數個
  // 標準型別（如int,c-style string 等）的hasher。
  hasher hash;
  key_equal equals;
  ExtractKey get_key;

  typedef __hashtable_node<Value> node;
  typedef simple_alloc<node, Alloc> node_allocator;

  vector<node*,Alloc> buckets;  // 以 vector 完成
  size_type num_elements;

public:
  typedef __hashtable_iterator<Value, Key, HashFcn, ExtractKey, EqualKey,
                               Alloc>
  iterator;

  typedef __hashtable_const_iterator<Value, Key, HashFcn, ExtractKey, EqualKey,
                                     Alloc>
  const_iterator;

  friend struct
  __hashtable_iterator<Value, Key, HashFcn, ExtractKey, EqualKey, Alloc>;
  friend struct
  __hashtable_const_iterator<Value, Key, HashFcn, ExtractKey, EqualKey, Alloc>;

public:
  // 注意，並沒有 default ctor
  hashtable(size_type n,
            const HashFcn&    hf,
            const EqualKey&   eql,
            const ExtractKey& ext)
    : hash(hf), equals(eql), get_key(ext), num_elements(0)
  {
    initialize_buckets(n);
  }

  hashtable(size_type n,
            const HashFcn&    hf,
            const EqualKey&   eql)
    : hash(hf), equals(eql), get_key(ExtractKey()), num_elements(0)
  {
    initialize_buckets(n);
  }

  hashtable(const hashtable& ht)
    : hash(ht.hash), equals(ht.equals), get_key(ht.get_key), num_elements(0)
  {
    copy_from(ht);
  }

  hashtable& operator= (const hashtable& ht)
```

```
{
  if (&ht != this) { // 標準的assignment op 判斷，避免自己指派給自己
    clear();         // 先清除自己
    hash = ht.hash;  // 以下三個動作，將三份data members 複製過來。
    equals = ht.equals;
    get_key = ht.get_key;
    copy_from(ht);   // 完整複製整個 hash table 的內容。
  }
  return *this;
}

~hashtable() { clear(); }

size_type size() const { return num_elements; }
size_type max_size() const { return size_type(-1); }
bool empty() const { return size() == 0; }

void swap(hashtable& ht)
{
  __STD::swap(hash, ht.hash);
  __STD::swap(equals, ht.equals);
  __STD::swap(get_key, ht.get_key);
  buckets.swap(ht.buckets);
  __STD::swap(num_elements, ht.num_elements);
}

iterator begin()
{
  for (size_type n = 0; n < buckets.size(); ++n)
    // 找出第一個被使用的節點，此即 begin iterator。
    if (buckets[n])
      return iterator(buckets[n], this);
  return end();
}

iterator end() { return iterator(0, this); }

const_iterator begin() const
{
  for (size_type n = 0; n < buckets.size(); ++n)
    if (buckets[n])
      return const_iterator(buckets[n], this);
  return end();
}

const_iterator end() const { return const_iterator(0, this); }

friend bool
operator== __STL_NULL_TMPL_ARGS (const hashtable&, const hashtable&);
```

```
public:

  // bucket 個數即 buckets vector 的大小
  size_type bucket_count() const { return buckets.size(); }

  // 以目前情況（不重建表格），總共可以有多少 buckets
  size_type max_bucket_count() const
    { return __stl_prime_list[__stl_num_primes - 1]; }

  // 探知某個bucket（內含一個list）容納多少元素。
  size_type elems_in_bucket(size_type bucket) const
  {
    size_type result = 0;
    for (node* cur = buckets[bucket]; cur; cur = cur->next)
      result += 1;
    return result;
  }

  // 安插元素，不允許重複
  pair<iterator, bool> insert_unique(const value_type& obj)
  {
    resize(num_elements + 1);    // 判斷是否需要重建表格，如需要就擴充
    return insert_unique_noresize(obj);
  }

  // 安插元素，允許重複
  iterator insert_equal(const value_type& obj)
  {
    resize(num_elements + 1);    // 判斷是否需要重建表格，如需要就擴充
    return insert_equal_noresize(obj);
  }

  pair<iterator, bool> insert_unique_noresize(const value_type& obj);
  iterator insert_equal_noresize(const value_type& obj);

#ifdef __STL_MEMBER_TEMPLATES
  template <class InputIterator>
  void insert_unique(InputIterator f, InputIterator l)
  {
    insert_unique(f, l, iterator_category(f));
  }

  template <class InputIterator>
  void insert_equal(InputIterator f, InputIterator l)
  {
    insert_equal(f, l, iterator_category(f));
  }
  template <class InputIterator>
  void insert_unique(InputIterator f, InputIterator l,
                  input_iterator_tag)
  {
    for ( ; f != l; ++f)
      insert_unique(*f);
  }

  template <class InputIterator>
  void insert_equal(InputIterator f, InputIterator l,
                 input_iterator_tag)
  {
    for ( ; f != l; ++f)
      insert_equal(*f);
  }

  template <class ForwardIterator>
  void insert_unique(ForwardIterator f, ForwardIterator l,
                  forward_iterator_tag)
  {
    size_type n = 0;
    distance(f, l, n);
    resize(num_elements + n);        // 判斷（並實施）表格的重建
    for ( ; n > 0; --n, ++f)
      insert_unique_noresize(*f);    // 一一安插新元素
  }

  template <class ForwardIterator>
  void insert_equal(ForwardIterator f, ForwardIterator l,
                 forward_iterator_tag)
  {
    size_type n = 0;
    distance(f, l, n);
    resize(num_elements + n);        // 判斷（並實施）表格的重建
    for ( ; n > 0; --n, ++f)
      insert_equal_noresize(*f);     // 一一安插新元素
  }

#else /* __STL_MEMBER_TEMPLATES */
  void insert_unique(const value_type* f, const value_type* l)
  {
    size_type n = l - f;
    resize(num_elements + n);
    for ( ; n > 0; --n, ++f)
      insert_unique_noresize(*f);
  }

  void insert_equal(const value_type* f, const value_type* l)
  {
    size_type n = l - f;
    resize(num_elements + n);
    for ( ; n > 0; --n, ++f)
      insert_equal_noresize(*f);
  }

  void insert_unique(const_iterator f, const_iterator l)
  {
    size_type n = 0;
    distance(f, l, n);
    resize(num_elements + n);
    for ( ; n > 0; --n, ++f)
      insert_unique_noresize(*f);
  }

  void insert_equal(const_iterator f, const_iterator l)
  {
    size_type n = 0;
    distance(f, l, n);
    resize(num_elements + n);
    for ( ; n > 0; --n, ++f)
      insert_equal_noresize(*f);
  }
#endif /*__STL_MEMBER_TEMPLATES */

  reference find_or_insert(const value_type& obj);

  iterator find(const key_type& key)
  {
    size_type n = bkt_num_key(key);  // 首先尋找落在哪一個 bucket 內
    node* first;
    // 以下，從bucket list 的頭開始，一一比對每個元素的鍵值。比對成功就跳出。
    for ( first = buckets[n];
          first && !equals(get_key(first->val), key);
          first = first->next)
      {}
    return iterator(first, this);
  }

  const_iterator find(const key_type& key) const
  {
    size_type n = bkt_num_key(key);
    const node* first;
    for ( first = buckets[n];
          first && !equals(get_key(first->val), key);
          first = first->next)
      {}
    return const_iterator(first, this);
  }
  size_type count(const key_type& key) const
  {
    const size_type n = bkt_num_key(key); // 首先尋找落在哪一個 bucket 內
    size_type result = 0;

    // 以下，從bucket list 的頭開始，一一比對每個元素的鍵值。比對成功就累加 1。
    for (const node* cur = buckets[n]; cur; cur = cur->next)
      if (equals(get_key(cur->val), key))
        ++result;
    return result;
  }

  pair<iterator, iterator> equal_range(const key_type& key);
  pair<const_iterator, const_iterator> equal_range(const key_type& key) const;

  size_type erase(const key_type& key);
  void erase(const iterator& it);
  void erase(iterator first, iterator last);

  void erase(const const_iterator& it);
  void erase(const_iterator first, const_iterator last);

  void resize(size_type num_elements_hint);
  void clear();

private:
  //  以下尋找STL 提供的下一個質數
  size_type next_size(size_type n) const { return __stl_next_prime(n); }

  // 注意，hash_vec 和 hash_map 都將其底層的 hash table 的初始大小預設為 100
  void initialize_buckets(size_type n)
  {
    const size_type n_buckets = next_size(n);
    // 例：傳入 100，傳回 193。以下首先保留 193 個元素空間，然後將其全部填 0。
    // 例：傳入 50，傳回 53。以下首先保留 53 個元素空間，然後將其全部填 0。
    buckets.reserve(n_buckets);
    buckets.insert(buckets.end(), n_buckets, (node*) 0);
    num_elements = 0;
  }

  size_type bkt_num_key(const key_type& key) const
  {
    return bkt_num_key(key, buckets.size());
  }

  size_type bkt_num(const value_type& obj) const
  {
    return bkt_num_key(get_key(obj));
  }

  size_type bkt_num_key(const key_type& key, size_t n) const
  {
    return hash(key) % n;
  }

  size_type bkt_num(const value_type& obj, size_t n) const
  {
    return bkt_num_key(get_key(obj), n);
  }

  node* new_node(const value_type& obj)
  {
    node* n = node_allocator::allocate();
    n->next = 0;
    __STL_TRY {
      construct(&n->val, obj);
      return n;
    }
    __STL_UNWIND(node_allocator::deallocate(n));
  }

  void delete_node(node* n)
  {
    destroy(&n->val);
    node_allocator::deallocate(n);
  }

  void erase_bucket(const size_type n, node* first, node* last);
  void erase_bucket(const size_type n, node* last);

  void copy_from(const hashtable& ht);

};

template <class V, class K, class HF, class ExK, class EqK, class A>
__hashtable_iterator<V, K, HF, ExK, EqK, A>&
__hashtable_iterator<V, K, HF, ExK, EqK, A>::operator++()
{
  const node* old = cur;
  cur = cur->next;     // 如果存在，就是它。否則進入以下 if 流程
  if (!cur) {
    // 根據原值，重新定位。從該位置（bucket）的下一位置找起。
    size_type bucket = ht->bkt_num(old->val);
    while (!cur && ++bucket < ht->buckets.size())  // 注意，prefix operator++
      cur = ht->buckets[bucket];
  }
  return *this;
```

```
}

template <class V, class K, class HF, class ExK, class EqK, class A>
inline __hashtable_iterator<V, K, HF, ExK, EqK, A>
__hashtable_iterator<V, K, HF, ExK, EqK, A>::operator++(int)
{
  iterator tmp = *this;
  ++*this;   // 喚起 operator++()
  return tmp;
}

template <class V, class K, class HF, class ExK, class EqK, class A>
__hashtable_const_iterator<V, K, HF, ExK, EqK, A>&
__hashtable_const_iterator<V, K, HF, ExK, EqK, A>::operator++()
{
  const node* old = cur;
  cur = cur->next;
  if (!cur) {
    size_type bucket = ht->bkt_num(old->val);
    while (!cur && ++bucket < ht->buckets.size())
      cur = ht->buckets[bucket];
  }
  return *this;
}

template <class V, class K, class HF, class ExK, class EqK, class A>
inline __hashtable_const_iterator<V, K, HF, ExK, EqK, A>
__hashtable_const_iterator<V, K, HF, ExK, EqK, A>::operator++(int)
{
  const_iterator tmp = *this;
  ++*this;
  return tmp;
}

#ifndef __STL_CLASS_PARTIAL_SPECIALIZATION

template <class V, class K, class HF, class ExK, class EqK, class All>
inline forward_iterator_tag
iterator_category(const __hashtable_iterator<V, K, HF, ExK, EqK, All>&)
{
  return forward_iterator_tag();
}

template <class V, class K, class HF, class ExK, class EqK, class All>
inline V* value_type(const __hashtable_iterator<V, K, HF, ExK, EqK, All>&)
{
  return (V*) 0;
}
```

```
template <class V, class K, class HF, class ExK, class EqK, class All>
inline hashtable<V, K, HF, ExK, EqK, All>::difference_type*
distance_type(const __hashtable_iterator<V, K, HF, ExK, EqK, All>&)
{
  return (hashtable<V, K, HF, ExK, EqK, All>::difference_type*) 0;
}

template <class V, class K, class HF, class ExK, class EqK, class All>
inline forward_iterator_tag
iterator_category(const __hashtable_const_iterator<V, K, HF, ExK, EqK, All>&)
{
  return forward_iterator_tag();
}

template <class V, class K, class HF, class ExK, class EqK, class All>
inline V*
value_type(const __hashtable_const_iterator<V, K, HF, ExK, EqK, All>&)
{
  return (V*) 0;
}

template <class V, class K, class HF, class ExK, class EqK, class All>
inline hashtable<V, K, HF, ExK, EqK, All>::difference_type*
distance_type(const __hashtable_const_iterator<V, K, HF, ExK, EqK, All>&)
{
  return (hashtable<V, K, HF, ExK, EqK, All>::difference_type*) 0;
}

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

template <class V, class K, class HF, class Ex, class Eq, class A>
bool operator==(const hashtable<V, K, HF, Ex, Eq, A>& ht1,
                const hashtable<V, K, HF, Ex, Eq, A>& ht2)
{
  typedef typename hashtable<V, K, HF, Ex, Eq, A>::node node;
  if (ht1.buckets.size() != ht2.buckets.size())
    return false;
  for (int n = 0; n < ht1.buckets.size(); ++n) {
    node* cur1 = ht1.buckets[n];
    node* cur2 = ht2.buckets[n];
    for ( ; cur1 && cur2 && cur1->val == cur2->val;
          cur1 = cur1->next, cur2 = cur2->next)
      {}
    if (cur1 || cur2)
      return false;
  }
  return true;
}
```

```
#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

template <class Val, class Key, class HF, class Extract, class EqKey, class A>
inline void swap(hashtable<Val, Key, HF, Extract, EqKey, A>& ht1,
              hashtable<Val, Key, HF, Extract, EqKey, A>& ht2) {
  ht1.swap(ht2);
}

#endif /* __STL_FUNCTION_TMPL_PARTIAL_ORDER */

// 在不需重建表格的情況下安插新節點。鍵值不允許重複。
template <class V, class K, class HF, class Ex, class Eq, class A>
pair<typename hashtable<V, K, HF, Ex, Eq, A>::iterator, bool>
hashtable<V, K, HF, Ex, Eq, A>::insert_unique_noresize(const value_type& obj)
{
  const size_type n = bkt_num(obj);  // 決定obj應位於 #n bucket
  node* first = buckets[n]; // 令 first 指向 bucket 對應之串列頭部

  // 如果 buckets[n] 已被佔用，此時first 將不為 0，於是進入以下迴圈，
  // 走過 bucket 所對應的整個串列。
  for (node* cur = first; cur; cur = cur->next)
    if (equals(get_key(cur->val), get_key(obj)))
      // 如果發現與串列中的某鍵值相同，就不安插，立刻回返。
      return pair<iterator, bool>(iterator(cur, this), false);

  // 離開以上迴圈（或根本未進入迴圈）時，first 指向bucket所指串列的頭部節點。
  node* tmp = new_node(obj);    // 產生新節點
  tmp->next = first;
  buckets[n] = tmp;             // 令新節點成為串列的第一個節點
  ++num_elements;               // 節點個數累加 1
  return pair<iterator, bool>(iterator(tmp, this), true);
}

// 在不需重建表格的情況下安插新節點。鍵值允許重複。
template <class V, class K, class HF, class Ex, class Eq, class A>
typename hashtable<V, K, HF, Ex, Eq, A>::iterator
hashtable<V, K, HF, Ex, Eq, A>::insert_equal_noresize(const value_type& obj)
{
  const size_type n = bkt_num(obj); // 決定obj應位於 #n bucket
  node* first = buckets[n]; // 令 first 指向 bucket 對應之串列頭部

  // 如果 buckets[n] 已被佔用，此時first 將不為 0，於是進入以下迴圈，
  // 走過 bucket 所對應的整個串列。
  for (node* cur = first; cur; cur = cur->next)
    if (equals(get_key(cur->val), get_key(obj))) {
      // 如果發現與串列中的某鍵值相同，就馬上安插，然後回返。
      node* tmp = new_node(obj);      // 產生新節點
      tmp->next = cur->next;          // 將新節點安插於目前位置
      cur->next = tmp;
      ++num_elements;                 // 節點個數累加 1
      return iterator(tmp, this);     // 傳回一個迭代器，指向新增節點
    }

  // 進行至此，表示沒有發現重複的鍵值
  node* tmp = new_node(obj);      // 產生新節點
  tmp->next = first;              // 將新節點安插於串列頭部
  buckets[n] = tmp;
  ++num_elements;                 // 節點個數累加 1
  return iterator(tmp, this);     // 傳回一個迭代器，指向新增節點
}

template <class V, class K, class HF, class Ex, class Eq, class A>
typename hashtable<V, K, HF, Ex, Eq, A>::reference
hashtable<V, K, HF, Ex, Eq, A>::find_or_insert(const value_type& obj)
{
  resize(num_elements + 1);

  size_type n = bkt_num(obj);
  node* first = buckets[n];

  for (node* cur = first; cur; cur = cur->next)
    if (equals(get_key(cur->val), get_key(obj)))
      return cur->val;

  node* tmp = new_node(obj);
  tmp->next = first;
  buckets[n] = tmp;
  ++num_elements;
  return tmp->val;
}

template <class V, class K, class HF, class Ex, class Eq, class A>
pair<typename hashtable<V, K, HF, Ex, Eq, A>::iterator,
     typename hashtable<V, K, HF, Ex, Eq, A>::iterator>
hashtable<V, K, HF, Ex, Eq, A>::equal_range(const key_type& key)
{
  typedef pair<iterator, iterator> pii;
  const size_type n = bkt_num_key(key);

  for (node* first = buckets[n]; first; first = first->next) {
    if (equals(get_key(first->val), key)) {
      for (node* cur = first->next; cur; cur = cur->next)
        if (!equals(get_key(cur->val), key))
          return pii(iterator(first, this), iterator(cur, this));
      for (size_type m = n + 1; m < buckets.size(); ++m)
        if (buckets[m])
          return pii(iterator(first, this),
                     iterator(buckets[m], this));
      return pii(iterator(first, this), end());
    }
  }
  return pii(end(), end());
}

template <class V, class K, class HF, class Ex, class Eq, class A>
pair<typename hashtable<V, K, HF, Ex, Eq, A>::const_iterator,
     typename hashtable<V, K, HF, Ex, Eq, A>::const_iterator>
hashtable<V, K, HF, Ex, Eq, A>::equal_range(const key_type& key) const
{
  typedef pair<const_iterator, const_iterator> pii;
  const size_type n = bkt_num_key(key);

  for (const node* first = buckets[n] ; first; first = first->next) {
    if (equals(get_key(first->val), key)) {
      for (const node* cur = first->next; cur; cur = cur->next)
        if (!equals(get_key(cur->val), key))
          return pii(const_iterator(first, this),
                     const_iterator(cur, this));
      for (size_type m = n + 1; m < buckets.size(); ++m)
        if (buckets[m])
          return pii(const_iterator(first, this),
                     const_iterator(buckets[m], this));
      return pii(const_iterator(first, this), end());
    }
  }
  return pii(end(), end());
}

template <class V, class K, class HF, class Ex, class Eq, class A>
typename hashtable<V, K, HF, Ex, Eq, A>::size_type
hashtable<V, K, HF, Ex, Eq, A>::erase(const key_type& key)
{
  const size_type n = bkt_num_key(key);
  node* first = buckets[n];
  size_type erased = 0;

  if (first) {
    node* cur = first;
    node* next = cur->next;
    while (next) {
      if (equals(get_key(next->val), key)) {
        cur->next = next->next;
        delete_node(next);
        next = cur->next;
        ++erased;
        --num_elements;
      }
```

```
   else {
    cur = next;
    next = cur->next;
   }
  }
  if (equals(get_key(first->val), key)) {
   buckets[n] = first->next;
   delete_node(first);
   ++erased;
   --num_elements;
  }
 }
 return erased;
}

template <class V, class K, class HF, class Ex, class Eq, class A>
void hashtable<V, K, HF, Ex, Eq, A>::erase(const iterator& it)
{
 if (node* const p = it.cur) {
  const size_type n = bkt_num(p->val);
  node* cur = buckets[n];

  if (cur == p) {
   buckets[n] = cur->next;
   delete_node(cur);
   --num_elements;
  }
  else {
   node* next = cur->next;
   while (next) {
    if (next == p) {
     cur->next = next->next;
     delete_node(next);
     --num_elements;
     break;
    }
    else {
     cur = next;
     next = cur->next;
    }
   }
  }
 }
}

template <class V, class K, class HF, class Ex, class Eq, class A>
void hashtable<V, K, HF, Ex, Eq, A>::erase(iterator first, iterator last)
{
 size_type f_bucket = first.cur ? bkt_num(first.cur->val) : buckets.size();
```

```
  size_type l_bucket = last.cur ? bkt_num(last.cur->val) : buckets.size();

  if (first.cur == last.cur)
    return;
  else if (f_bucket == l_bucket)
    erase_bucket(f_bucket, first.cur, last.cur);
  else {
    erase_bucket(f_bucket, first.cur, 0);
    for (size_type n = f_bucket + 1; n < l_bucket; ++n)
      erase_bucket(n, 0);
    if (l_bucket != buckets.size())
      erase_bucket(l_bucket, last.cur);
  }
}

template <class V, class K, class HF, class Ex, class Eq, class A>
inline void
hashtable<V, K, HF, Ex, Eq, A>::erase(const_iterator first,
                                      const_iterator last)
{
  erase(iterator(const_cast<node*>(first.cur),
                 const_cast<hashtable*>(first.ht)),
        iterator(const_cast<node*>(last.cur),
                 const_cast<hashtable*>(last.ht)));
}

template <class V, class K, class HF, class Ex, class Eq, class A>
inline void
hashtable<V, K, HF, Ex, Eq, A>::erase(const const_iterator& it)
{
  erase(iterator(const_cast<node*>(it.cur),
                 const_cast<hashtable*>(it.ht)));
}

template <class V, class K, class HF, class Ex, class Eq, class A>
void hashtable<V, K, HF, Ex, Eq, A>::resize(size_type num_elements_hint)
{
  const size_type old_n = buckets.size();
  if (num_elements_hint > old_n) {   // 確定真的需要重新配置
    const size_type n = next_size(num_elements_hint);   // 找出下一個質數
    if (n > old_n) {
      vector<node*, A> tmp(n, (node*) 0); // 設立新的 buckets
      __STL_TRY {
        // 以下處理每一個舊的bucket
        for (size_type bucket = 0; bucket < old_n; ++bucket) {
          node* first = buckets[bucket]; // 指向節點所對應之串列的起始節點
          // 以下處理每一個舊bucket 所含（串列）的每一個節點
          while (first) {    // 串列還沒結束時
            // 以下找出節點落在哪一個新bucket 內
            size_type new_bucket = bkt_num(first->val, n);
            // 以下四個動作頗為微妙
            // (1) 令舊 bucket 指向其所對應之串列的下一個節點（以便迭代處理）
            buckets[bucket] = first->next;
            // (2)(3) 將當前節點安插到新bucket 內，成為其對應串列的第一個節點。
            first->next = tmp[new_bucket];
            tmp[new_bucket] = first;
            // (4) 回到舊bucket 所指的待處理串列，準備處理下一個節點
            first = buckets[bucket];
          }
        }
        buckets.swap(tmp);  // vector::swap。新舊 buckets 對調。
        // 注意，對調兩方如果大小不同，大的會變小，小的會變大。
        // 離開時釋還local tmp的記憶體。
      }
#       ifdef __STL_USE_EXCEPTIONS
      catch(...) {
        for (size_type bucket = 0; bucket < tmp.size(); ++bucket) {
          while (tmp[bucket]) {
            node* next = tmp[bucket]->next;
            delete_node(tmp[bucket]);
            tmp[bucket] = next;
          }
        }
        throw;
      }
#       endif /* __STL_USE_EXCEPTIONS */
    }
  }
}

template <class V, class K, class HF, class Ex, class Eq, class A>
void hashtable<V, K, HF, Ex, Eq, A>::erase_bucket(const size_type n,
                                                  node* first, node* last)
{
  node* cur = buckets[n];
  if (cur == first)
    erase_bucket(n, last);
  else {
    node* next;
    for (next = cur->next; next != first; cur = next, next = cur->next)
      ;
    while (next) {
      cur->next = next->next;
      delete_node(next);
      next = cur->next;
      --num_elements;
    }
  }
```

```
}

template <class V, class K, class HF, class Ex, class Eq, class A>
void
hashtable<V, K, HF, Ex, Eq, A>::erase_bucket(const size_type n, node* last)
{
  node* cur = buckets[n];
  while (cur != last) {
    node* next = cur->next;
    delete_node(cur);
    cur = next;
    buckets[n] = cur;
    --num_elements;
  }
}

template <class V, class K, class HF, class Ex, class Eq, class A>
void hashtable<V, K, HF, Ex, Eq, A>::clear()
{
  // 針對每一個 bucket.
  for (size_type i = 0; i < buckets.size(); ++i) {
    node* cur = buckets[i];
    // 將 bucket list 中的每一個節點刪除掉
    while (cur != 0) {
      node* next = cur->next;
      delete_node(cur);
      cur = next;
    }
    buckets[i] = 0;    // 令bucket 內容為 null 指標
  }
  num_elements = 0;    // 令總節點個數為 0

  // 注意，buckets vector 並未釋放掉空間，仍保有原來大小。
}


template <class V, class K, class HF, class Ex, class Eq, class A>
void hashtable<V, K, HF, Ex, Eq, A>::copy_from(const hashtable& ht)
{
  // 先清除己方的buckets vector. 這動作是呼叫vector::clear. 造成所有元素為 0
  buckets.clear();
  // 為己方的buckets vector 保留空間，使與對方相同
  // 如果己方空間大於對方，就不動，如果己方空間小於對方，就會增大。
  buckets.reserve(ht.buckets.size());
  // 從己方的 buckets vector 尾端開始，安插n個元素，其值為 null 指標。
  // 注意，此時buckets vector 為空，所以所謂尾端，就是起頭處。
  buckets.insert(buckets.end(), ht.buckets.size(), (node*) 0);
  __STL_TRY {
    // 針對 buckets vector
    for (size_type i = 0; i < ht.buckets.size(); ++i) {
      // 複製 vector 的每一個元素（是個指標，指向 hashtable節點）
      if (const node* cur = ht.buckets[i]) {
        node* copy = new_node(cur->val);
        buckets[i] = copy;

        // 針對同一個 bucket list，複製每一個節點
        for (node* next = cur->next; next; cur = next, next = cur->next) {
          copy->next = new_node(next->val);
          copy = copy->next;
        }
      }
    }
    num_elements = ht.num_elements;  // 重新登錄節點個數（hashtable 的大小）
  }
  __STL_UNWIND(clear());
}

__STL_END_NAMESPACE

#endif /* __SGI_STL_INTERNAL_HASHTABLE_H */

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