

```
G++ 2.91.57, cygnus\cygwin-b20\include\g++\stl_hash_set.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_SET_H
#define __SGI_STL_INTERNAL_HASH_SET_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 Value, class HashFcn = hash<Value>,
          class EqualKey = equal_to<Value>,
          class Alloc = alloc>
#else
template <class Value, class HashFcn, class EqualKey, class Alloc = alloc>
#endif
```

```

class hash_set
{
private:
    // 以下使用的 identity<> 定義於 <stl_function.h> 中。
    typedef hashtable<Value, Value, HashFcn, identity<Value>,
        EqualKey, Alloc> ht;
    ht rep;    // 底層機制以 hash table 完成

public:
    typedef typename ht::key_type key_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::const_pointer pointer;
    typedef typename ht::const_pointer const_pointer;
    typedef typename ht::const_reference reference;
    typedef typename ht::const_reference const_reference;

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

    hasher hash_func() const { return rep.hash_func(); }
    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_set() : rep(100, hasher(), key_equal()) {}
    explicit hash_set(size_type n) : rep(n, hasher(), key_equal()) {}
    hash_set(size_type n, const hasher& hf) : rep(n, hf, key_equal()) {}
    hash_set(size_type n, const hasher& hf, const key_equal& eql)
        : rep(n, hf, eql) {}

    // 以下，安插動作全部使用 insert_unique()，不允許鍵值重複。
#ifdef __STL_MEMBER_TEMPLATES
    template <class InputIterator>
    hash_set(InputIterator f, InputIterator l)
        : rep(100, hasher(), key_equal()) { rep.insert_unique(f, l); }
    template <class InputIterator>
    hash_set(InputIterator f, InputIterator l, size_type n)
        : rep(n, hasher(), key_equal()) { rep.insert_unique(f, l); }
    template <class InputIterator>
    hash_set(InputIterator f, InputIterator l, size_type n,
        const hasher& hf)
        : rep(n, hf, key_equal()) { rep.insert_unique(f, l); }
    template <class InputIterator>
    hash_set(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_set(const value_type* f, const value_type* l)
        : rep(100, hasher(), key_equal()) { rep.insert_unique(f, l); }
    hash_set(const value_type* f, const value_type* l, size_type n)
        : rep(n, hasher(), key_equal()) { rep.insert_unique(f, l); }
    hash_set(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_set(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_set(const_iterator f, const_iterator l)
        : rep(100, hasher(), key_equal()) { rep.insert_unique(f, l); }
    hash_set(const_iterator f, const_iterator l, size_type n)
        : rep(n, hasher(), key_equal()) { rep.insert_unique(f, l); }
    hash_set(const_iterator f, const_iterator l, size_type n,
              const hasher& hf)
        : rep(n, hf, key_equal()) { rep.insert_unique(f, l); }
    hash_set(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_set& hs) { rep.swap(hs.rep); }
    friend bool operator== (__STL_NULL_TMPL_ARGS (const hash_set&,
                                                    const hash_set&));

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

public:
    pair<iterator, bool> insert(const value_type& obj)
    {
        pair<typename ht::iterator, bool> p = rep.insert_unique(obj);
        return pair<iterator, bool>(p.first, p.second);
    }
    #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)
{
    pair<typename ht::iterator, bool> p = rep.insert_unique_noresize(obj);
    return pair<iterator, bool>(p.first, p.second);
}

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) 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 Value, class HashFcn, class EqualKey, class Alloc>
inline bool operator==(const hash_set<Value, HashFcn, EqualKey, Alloc>& hs1,
                      const hash_set<Value, HashFcn, EqualKey, Alloc>& hs2)
{
    return hs1.rep == hs2.rep;
}

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

template <class Val, class HashFcn, class EqualKey, class Alloc>
inline void swap(hash_set<Val, HashFcn, EqualKey, Alloc>& hs1,
                hash_set<Val, HashFcn, EqualKey, Alloc>& hs2) {
    hs1.swap(hs2);
}

#endif /* __STL_FUNCTION_TMPL_PARTIAL_ORDER */

```

```

#ifndef __STL_LIMITED_DEFAULT_TEMPLATES
template <class Value, class HashFcn = hash<Value>,
         class EqualKey = equal_to<Value>,
         class Alloc = alloc>
#else
template <class Value, class HashFcn, class EqualKey, class Alloc = alloc>
#endif
class hash_multiset
{
private:
    typedef hashtable<Value, Value, HashFcn, identity<Value>,
                    EqualKey, Alloc> ht;
    ht rep;

public:
    typedef typename ht::key_type key_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::const_pointer pointer;
    typedef typename ht::const_pointer const_pointer;
    typedef typename ht::const_reference reference;
    typedef typename ht::const_reference const_reference;

    typedef typename ht::const_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_multiset() : rep(100, hasher(), key_equal()) {}
    explicit hash_multiset(size_type n) : rep(n, hasher(), key_equal()) {}
    hash_multiset(size_type n, const hasher& hf) : rep(n, hf, key_equal()) {}
    hash_multiset(size_type n, const hasher& hf, const key_equal& eql)
        : rep(n, hf, eql) {}

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

```

```

template <class InputIterator>
hash_multiset(InputIterator f, InputIterator l, size_type n,
              const hasher& hf)
    : rep(n, hf, key_equal()) { rep.insert_equal(f, l); }
template <class InputIterator>
hash_multiset(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_multiset(const value_type* f, const value_type* l)
    : rep(100, hasher(), key_equal()) { rep.insert_equal(f, l); }
hash_multiset(const value_type* f, const value_type* l, size_type n)
    : rep(n, hasher(), key_equal()) { rep.insert_equal(f, l); }
hash_multiset(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_multiset(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_multiset(const_iterator f, const_iterator l)
    : rep(100, hasher(), key_equal()) { rep.insert_equal(f, l); }
hash_multiset(const_iterator f, const_iterator l, size_type n)
    : rep(n, hasher(), key_equal()) { rep.insert_equal(f, l); }
hash_multiset(const_iterator f, const_iterator l, size_type n,
              const hasher& hf)
    : rep(n, hf, key_equal()) { rep.insert_equal(f, l); }
hash_multiset(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_multiset& hs) { rep.swap(hs.rep); }
    friend bool operator== (__STL_NULL_TMPL_ARGS (const hash_multiset&,
                                                const hash_multiset&);

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

public:
    iterator insert(const value_type& obj) { return rep.insert_equal(obj); }
#endif /*__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) 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) 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 Val, class HashFcn, class EqualKey, class Alloc>
inline bool operator==(const hash_multiset<Val, HashFcn, EqualKey, Alloc>& hs1,
                      const hash_multiset<Val, HashFcn, EqualKey, Alloc>& hs2)
{
    return hs1.rep == hs2.rep;
}

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

template <class Val, class HashFcn, class EqualKey, class Alloc>
inline void swap(hash_multiset<Val, HashFcn, EqualKey, Alloc>& hs1,
                 hash_multiset<Val, HashFcn, EqualKey, Alloc>& hs2)
{
    hs1.swap(hs2);
}

#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_SET_H */

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