G++ 2.91.57，cygnus\cygwin-b20\include\g++\stl_map.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_MAP_H
#define __SGI_STL_INTERNAL_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
// 注意，以下Key 為鍵值（key）型別，T為資料（data）型別。
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 map {
public:
```

```
// typedefs:

  typedef Key key_type;    // 鍵值型別
  typedef T data_type;     // 資料（真值）型別
  typedef T mapped_type;   //
  typedef pair<const Key, T> value_type; // 元素型別（鍵值/真值）
  typedef Compare key_compare;  // 鍵值比較函式

  // 以下定義一個 functor，其作用就是喚起 元素比較函式。
  class value_compare
    : public binary_function<value_type, value_type, bool> {
  friend class map<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:
  // 以下定義表述型別（representation type）。以map元素型別（一個pair）
  // 的第一型別，做為RB-tree節點的鍵值型別。
  typedef rb_tree<key_type, value_type,
               select1st<value_type>, key_compare, Alloc> rep_type;
  rep_type t;  // 以紅黑樹（RB-tree）表現 map
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;
  // 注意上一行，為什麼不像set一樣地將iterator 定義為 RB-tree 的 const_iterator？
  // 按說map 的元素有一定次序安排，不允許使用者在任意處做寫入動作，因此
  // 迭代器應該無法執行寫入動作才是。
  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
  // 注意， map 一定使用 insert_unique() 而不使用 insert_equal()。
  // multimap 才使用 insert_equal()。

  map() : t(Compare()) {}
  explicit map(const Compare& comp) : t(comp) {}
```

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

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

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

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

  // accessors:
  // 以下所有的 map操作行為，RB-tree 都已提供，所以map只要轉呼叫即可。

  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(); }
  // 注意以下 註標（subscript）運算子
  T& operator[](const key_type& k) {
    return (*((insert(value_type(k, T()))).first)).second;
  }
  void swap(map<Key, T, Compare, Alloc>& x) { t.swap(x.t); }
  // insert/erase

  // 注意以下 insert 動作傳回的型別
  pair<iterator,bool> insert(const value_type& x) { return t.insert_unique(x); }
  iterator insert(iterator position, const value_type& x) {
    return t.insert_unique(position, x);
  }
#ifdef __STL_MEMBER_TEMPLATES
  template <class InputIterator>
  void insert(InputIterator first, InputIterator last) {
    t.insert_unique(first, last);
  }
#else
  void insert(const value_type* first, const value_type* last) {
    t.insert_unique(first, last);
  }
  void insert(const_iterator first, const_iterator last) {
    t.insert_unique(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(); }

  // map 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 map&, const map&);
  friend bool operator< __STL_NULL_TMPL_ARGS (const map&, const map&);
```

```
};

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

template <class Key, class T, class Compare, class Alloc>
inline bool operator<(const map<Key, T, Compare, Alloc>& x,
                      const map<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(map<Key, T, Compare, Alloc>& x,
                 map<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_MAP_H */

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