```
G++ 2.91.57，cygnus\cygwin-b20\include\g++\stl_queue.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_QUEUE_H
#define __SGI_STL_INTERNAL_QUEUE_H

__STL_BEGIN_NAMESPACE

#ifndef __STL_LIMITED_DEFAULT_TEMPLATES
template <class T, class Sequence = deque<T> >
#else
template <class T, class Sequence>
#endif
class queue {
  friend bool operator== __STL_NULL_TMPL_ARGS (const queue& x, const queue& y);
  friend bool operator< __STL_NULL_TMPL_ARGS (const queue& x, const queue& y);
public:
  typedef typename Sequence::value_type value_type;
  typedef typename Sequence::size_type size_type;
  typedef typename Sequence::reference reference;
  typedef typename Sequence::const_reference const_reference;
```

```
protected:
  Sequence c;     // 底層容器
public:
  // 以下完全利用 Sequence c 的操作，完成 queue 的操作。
  bool empty() const { return c.empty(); }
  size_type size() const { return c.size(); }
  reference front() { return c.front(); }
  const_reference front() const { return c.front(); }
  reference back() { return c.back(); }
  const_reference back() const { return c.back(); }
  // deque 是兩頭可進出，queue 是末端進，前端出（所以先進者先出）。
  void push(const value_type& x) { c.push_back(x); }
  void pop() { c.pop_front(); }
};

template <class T, class Sequence>
bool operator==(const queue<T, Sequence>& x, const queue<T, Sequence>& y)
{
  return x.c == y.c;
}

template <class T, class Sequence>
bool operator<(const queue<T, Sequence>& x, const queue<T, Sequence>& y)
{
  return x.c < y.c;
}

#ifndef __STL_LIMITED_DEFAULT_TEMPLATES
/*
預設情況下 priority_queue 係利用 vector 完成一個 max-heap，後者乃為一個以
array（或 vector）表現的二元樹，其條件是，必須為完全樹（complete tree，此為
結構特性），且每個節點值都大於或等於其任一子節點值（此為次序特性）。因此根節點為
最大值。Max-heap適用於 priority_queue 所需特性。
*/
template <class T, class Sequence = vector<T>,
          class Compare = less<typename Sequence::value_type> >
#else
template <class T, class Sequence, class Compare>
#endif
class  priority_queue {
public:
  typedef typename Sequence::value_type value_type;
  typedef typename Sequence::size_type size_type;
  typedef typename Sequence::reference reference;
  typedef typename Sequence::const_reference const_reference;
protected:
  Sequence c;     // 底層容器
  Compare comp;   // 元素大小比較標準
public:
  priority_queue() : c() {}
  explicit priority_queue(const Compare& x) :  c(), comp(x) {}

// 以下用到的make_heap(), push_heap(), pop_heap()都是泛型演算法
// 注意，任一個建構式都立刻於底層容器內產生一個implicit representation heap。
#ifdef __STL_MEMBER_TEMPLATES
  template <class InputIterator>
  priority_queue(InputIterator first, InputIterator last, const Compare& x)
    : c(first, last), comp(x) { make_heap(c.begin(), c.end(), comp); }
  template <class InputIterator>
  priority_queue(InputIterator first, InputIterator last)
    : c(first, last) { make_heap(c.begin(), c.end(), comp); }
#else /* __STL_MEMBER_TEMPLATES */
  priority_queue(const value_type* first, const value_type* last,
                 const Compare& x) : c(first, last), comp(x) {
    make_heap(c.begin(), c.end(), comp);
  }
  priority_queue(const value_type* first, const value_type* last)
    : c(first, last) { make_heap(c.begin(), c.end(), comp); }
#endif /* __STL_MEMBER_TEMPLATES */

  bool empty() const { return c.empty(); }
  size_type size() const { return c.size(); }
  const_reference top() const { return c.front(); }
  void push(const value_type& x) {
    __STL_TRY {
      // push_heap 是泛型演算法，先利用底層容器的 push_back() 將新元素
      // 推入末端，再重排 heap。見C++ Primer p.1195。
      c.push_back(x);
      push_heap(c.begin(), c.end(), comp); // push_heap 是泛型演算法
    }
    __STL_UNWIND(c.clear());
  }
  void pop() {
    __STL_TRY {
      // pop_heap 是泛型演算法，從 heap 內取出一個元素。它並不是真正將元素
      // 彈出，而是重排 heap，然後再以底層容器的 pop_back() 取得被彈出
      // 的元素。見C++ Primer p.1195。
      pop_heap(c.begin(), c.end(), comp);
      c.pop_back();
    }
    __STL_UNWIND(c.clear());
  }
};

// no equality is provided

__STL_END_NAMESPACE
```

```
#endif /* __SGI_STL_INTERNAL_QUEUE_H */

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