G++ 2.91.57，cygnus\cygwin-b20\include\g++**stl_list.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_LIST_H
#define __SGI_STL_INTERNAL_LIST_H

__STL_BEGIN_NAMESPACE

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

// 串列節點結構。這是一個雙向串列
template <class T>
struct __list_node {
  typedef void* void_pointer;
  void_pointer next;  // 型別為 void*。其實可設為 __list_node<T>*
  void_pointer prev;
  T data;
```

```
};

// 串列專屬迭代器。既然撰寫串列迭代器避免不了要曝露串列的實作細節，
// 那麼就讓串列和串列迭代器一起設計好了。
template<class T, class Ref, class Ptr>
struct __list_iterator { // 未繼承 std::iterator
  typedef __list_iterator<T, T&, T*>           iterator;
  typedef __list_iterator<T, const T&, const T*> const_iterator;
  typedef __list_iterator<T, Ref, Ptr>         self;

  // 未繼承 std::iterator，所以必須自行撰寫五個必要的迭代器相應型別
  typedef bidirectional_iterator_tag iterator_category; // (1)
  typedef T value_type;              // (2)
  typedef Ptr pointer;               // (3)
  typedef Ref reference;             // (4)
  typedef __list_node<T>* link_type;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type; // (5)

  link_type node;  // 保持與容器的聯結

  // 以下 ctor 如有參數，便根據參數設定迭代器與容器之間的聯結關係
  __list_iterator(link_type x) : node(x) {}
  __list_iterator() {}
  __list_iterator(const iterator& x) : node(x.node) {}

  // 迭代器必要的操作行為
  bool operator==(const self& x) const { return node == x.node; }
  bool operator!=(const self& x) const { return node != x.node; }
  // 關鍵：對迭代器取值（dereference），取的是節點的資料值。
  reference operator*() const { return (*node).data; }

#ifndef __SGI_STL_NO_ARROW_OPERATOR
  pointer operator->() const { return &(operator*()); }
#endif /* __SGI_STL_NO_ARROW_OPERATOR */

  // 參考 More Effective C++, item6: Distinguish between prefix and
  // postfix forms of increment and decrement operators.
  // 關鍵：對迭代器累加 1，就是前進一個節點
  self& operator++() {
    node = (link_type)((*node).next);
    return *this;
  }
  self operator++(int) {
    self tmp = *this;
    ++*this;
    return tmp;
  }
  // 對迭代器累減 1，就是後退一個節點
```

```
  self& operator--() {
    node = (link_type)((*node).prev);     // 關鍵
    return *this;
  }
  self operator--(int) {
    self tmp = *this;
    --*this;
    return tmp;
  }
};

#ifndef __STL_CLASS_PARTIAL_SPECIALIZATION
// 編譯器不支援 partial specialization 時，才需以下定義
template <class T, class Ref, class Ptr>
inline bidirectional_iterator_tag
iterator_category(const __list_iterator<T, Ref, Ptr>&) {
  return bidirectional_iterator_tag();
}

template <class T, class Ref, class Ptr>
inline T*
value_type(const __list_iterator<T, Ref, Ptr>&) {
  return 0;
}

template <class T, class Ref, class Ptr>
inline ptrdiff_t*
distance_type(const __list_iterator<T, Ref, Ptr>&) {
  return 0;
}
// 編譯器不支援 partial specialization 時，才需以上定義
#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

template <class T, class Alloc = alloc> // 預設使用 alloc 為配置器
class list {
protected:
  typedef void* void_pointer;
  typedef __list_node<T> list_node;
  // 專屬之空間配置器，每次配置一個節點大小
  typedef simple_alloc<list_node, Alloc> list_node_allocator;
public:
  typedef T value_type;
  typedef value_type* pointer;
  typedef const value_type* const_pointer;
  typedef value_type& reference;
  typedef const value_type& const_reference;
  typedef list_node* link_type;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;
```

```
public:
  // 當客端定義一個 list<T>::iterator 物件，例如 list<T>::iterator，
  // 便喚起 __list_iterator<T, T&, T*> 的 ctor。
  // 如果有初值，便會因此設定一個迭代器對容器的聯結關係。
  typedef __list_iterator<T, T&, T*>           iterator;
  typedef __list_iterator<T, const T&, const T*> const_iterator;

#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION
  typedef reverse_iterator<const_iterator> const_reverse_iterator;
  typedef reverse_iterator<iterator> reverse_iterator;
#else /* __STL_CLASS_PARTIAL_SPECIALIZATION */
  typedef reverse_bidirectional_iterator<const_iterator, value_type,
  const_reference, difference_type>
  const_reverse_iterator;
  typedef reverse_bidirectional_iterator<iterator, value_type, reference,
  difference_type>
  reverse_iterator;
#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

protected:
  // 配置一個節點並傳回
  link_type get_node() { return list_node_allocator::allocate(); }
  // 釋放一個節點
  void put_node(link_type p) { list_node_allocator::deallocate(p); }

  // 產生（配置並建構）一個節點，帶有元素值
  link_type create_node(const T& x) {
    link_type p = get_node();
    __STL_TRY {
      construct(&p->data, x);    // 全域函式，建構/解構基本工具。
    }
    __STL_UNWIND(put_node(p));
    return p;
  }
  // 摧毀（解構並釋放）一個節點
  void destroy_node(link_type p) {
    destroy(&p->data);           // 全域函式，建構/解構基本工具。
    put_node(p);
  }

protected:
  void empty_initialize() {
    node = get_node(); // 配置一個節點空間，令 node 指向它。
    node->next = node; // 令node 頭尾都指向自己，不設元素值。
    node->prev = node;
  }

  void fill_initialize(size_type n, const T& value) {
```

```
    empty_initialize();
    __STL_TRY {
      insert(begin(), n, value);
    }
    __STL_UNWIND(clear(); put_node(node));
  }

#ifdef __STL_MEMBER_TEMPLATES
  template <class InputIterator>
  void range_initialize(InputIterator first, InputIterator last) {
    empty_initialize();
    __STL_TRY {
      insert(begin(), first, last);
    }
    __STL_UNWIND(clear(); put_node(node));
  }
#else  /* __STL_MEMBER_TEMPLATES */
  void range_initialize(const T* first, const T* last) {
    empty_initialize();
    __STL_TRY {
      insert(begin(), first, last);
    }
    __STL_UNWIND(clear(); put_node(node));
  }
  void range_initialize(const_iterator first, const_iterator last) {
    empty_initialize();
    __STL_TRY {
      insert(begin(), first, last);
    }
    __STL_UNWIND(clear(); put_node(node));
  }
#endif /* __STL_MEMBER_TEMPLATES */

protected:
  // 從實作細節看來，本 list 只維護一個節點指標，指向最後（尾）節點的下一位置。
  // 由於這是一個環狀雙向串列，因此，欲對外供應頭節點或尾節點，都十分容易，
  // 見 front(), back()。
  link_type node; // 永遠指向最後節點的下一節點。該節點無元素值，代表空節點。
                  // 其 next 節點永遠是頭節點。

public:
  list() { empty_initialize(); }  // 產生一個空串列。

  iterator begin() { return (link_type)((*node).next); }
  const_iterator begin() const { return (link_type)((*node).next); }
  // node 指向尾節點的下一位置，因此 node 符合STL對 end 的定義。
  iterator end() { return node; }
    const_iterator end() const { return node; }
  reverse_iterator rbegin() { return reverse_iterator(end()); }
```

```
  const_reverse_iterator rbegin() const {
    return const_reverse_iterator(end());
  }
  reverse_iterator rend() { return reverse_iterator(begin()); }
  const_reverse_iterator rend() const {
    return const_reverse_iterator(begin());
  }
  bool empty() const { return node->next == node; }
  size_type size() const {
    size_type result = 0;
    distance(begin(), end(), result);  // 全域函式，定義於 <stl_iterator.h>
    return result;
  }
  size_type max_size() const { return size_type(-1); }
  // 取頭節點的內容（元素值）。
  reference front() { return *begin(); }
  const_reference front() const { return *begin(); }
  // 取尾節點的內容（元素值）。
  reference back() { return *(--end()); }
  const_reference back() const { return *(--end()); }
  void swap(list<T, Alloc>& x) { __STD::swap(node, x.node); }

  // 在迭代器 position 所指位置安插一個節點，內容為 x。
  iterator insert(iterator position, const T& x) {
    link_type tmp = create_node(x); // 產生一個節點（設妥內容為 x）
    // 調整雙向指標，使 tmp 安插進去。
    tmp->next = position.node;
    tmp->prev = position.node->prev;
    (link_type(position.node->prev))->next = tmp;
    position.node->prev = tmp;
    return tmp;
  }
  iterator insert(iterator position) { return insert(position, T()); }
#ifdef __STL_MEMBER_TEMPLATES
  template <class InputIterator>
  void insert(iterator position, InputIterator first, InputIterator last);
#else /* __STL_MEMBER_TEMPLATES */
  void insert(iterator position, const T* first, const T* last);
  void insert(iterator position,
              const_iterator first, const_iterator last);
#endif /* __STL_MEMBER_TEMPLATES */
  void insert(iterator pos, size_type n, const T& x);
  void insert(iterator pos, int n, const T& x) {
    insert(pos, (size_type)n, x);
  }
  void insert(iterator pos, long n, const T& x) {
    insert(pos, (size_type)n, x);
  }
```

```
  // 安插一個節點，做為頭節點
  void push_front(const T& x) { insert(begin(), x); }
  // 安插一個節點，做為尾節點
  void push_back(const T& x) { insert(end(), x); }

  // 移除迭代器 position 所指節點
  iterator erase(iterator position) {
    link_type next_node = link_type(position.node->next);
    link_type prev_node = link_type(position.node->prev);
    prev_node->next = next_node;
    next_node->prev = prev_node;
    destroy_node(position.node);
    return iterator(next_node);
  }
  iterator erase(iterator first, iterator last);
  void resize(size_type new_size, const T& x);
  void resize(size_type new_size) { resize(new_size, T()); }
  void clear();

  // 移除頭節點
  void pop_front() { erase(begin()); }
  // 移除尾節點
  void pop_back() {
    iterator tmp = end();
    erase(--tmp);
  }
  list(size_type n, const T& value) { fill_initialize(n, value); }
  list(int n, const T& value) { fill_initialize(n, value); }
  list(long n, const T& value) { fill_initialize(n, value); }
  explicit list(size_type n) { fill_initialize(n, T()); }

#ifdef __STL_MEMBER_TEMPLATES
  template <class InputIterator>
  list(InputIterator first, InputIterator last) {
    range_initialize(first, last);
  }

#else /* __STL_MEMBER_TEMPLATES */
  list(const T* first, const T* last) { range_initialize(first, last); }
  list(const_iterator first, const_iterator last) {
    range_initialize(first, last);
  }
#endif /* __STL_MEMBER_TEMPLATES */
  list(const list<T, Alloc>& x) {
    range_initialize(x.begin(), x.end());
  }
  ~list() {
    clear();
    put_node(node);
```

```
  }
  list<T, Alloc>& operator=(const list<T, Alloc>& x);

protected:
  // 將 [first,last) 內的所有元素搬移到position 處。
  void transfer(iterator position, iterator first, iterator last) {
    if (position != last) {
      (*(link_type((*last.node).prev))).next = position.node;     // (1)
      (*(link_type((*first.node).prev))).next = last.node;        // (2)
      (*(link_type((*position.node).prev))).next = first.node;    // (3)
      link_type tmp = link_type((*position.node).prev);           // (4)
      (*position.node).prev = (*last.node).prev;                  // (5)
      (*last.node).prev = (*first.node).prev;                     // (6)
      (*first.node).prev = tmp;                                   // (7)
    }
  }

public:
  // 將 x 接合於 position 所指位置之前。x 必須不同於 *this。
  void splice(iterator position, list& x) {
    if (!x.empty())
      transfer(position, x.begin(), x.end());
  }
  // 將 i 所指元素接合於 position 所指位置之前。position 和i 可指向同一個list。
  void splice(iterator position, list&, iterator i) {
    iterator j = i;
    ++j;
    if (position == i || position == j) return;
    transfer(position, i, j);
  }
  // 將 [first,last) 內的所有元素接合於 position 所指位置之前。
  // position 和[first,last)可指向同一個list，
  // 但position不能位於[first,last)之內。
  void splice(iterator position, list&, iterator first, iterator last)  {
    if (first != last)
      transfer(position, first, last);
  }
  void remove(const T& value);
  void unique();
  void merge(list& x);
  void reverse();
  void sort();

#ifdef __STL_MEMBER_TEMPLATES
  template <class Predicate> void remove_if(Predicate);
  template <class BinaryPredicate> void unique(BinaryPredicate);
  template <class StrictWeakOrdering> void merge(list&, StrictWeakOrdering);
  template <class StrictWeakOrdering> void sort(StrictWeakOrdering);
#endif /* __STL_MEMBER_TEMPLATES */
```

```
  friend bool operator== __STL_NULL_TMPL_ARGS (const list& x, const list& y);
};

template <class T, class Alloc>
inline bool operator==(const list<T,Alloc>& x, const list<T,Alloc>& y)
{
  typedef typename list<T,Alloc>::link_type link_type;
  link_type e1 = x.node;
  link_type e2 = y.node;
  link_type n1 = (link_type) e1->next;
  link_type n2 = (link_type) e2->next;
  for ( ; n1 != e1 && n2 != e2 ;
        n1 = (link_type) n1->next, n2 = (link_type) n2->next)
    if (n1->data != n2->data)
      return false;
  return n1 == e1 && n2 == e2;
}

template <class T, class Alloc>
inline bool operator<(const list<T, Alloc>& x, const list<T, Alloc>& y) {
  return lexicographical_compare(x.begin(), x.end(), y.begin(), y.end());
}

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

template <class T, class Alloc>
inline void swap(list<T, Alloc>& x, list<T, Alloc>& y) {
  x.swap(y);
}

#endif /* __STL_FUNCTION_TMPL_PARTIAL_ORDER */

#ifdef __STL_MEMBER_TEMPLATES

template <class T, class Alloc> template <class InputIterator>
void list<T, Alloc>::insert(iterator position,
                            InputIterator first, InputIterator last) {
  for ( ; first != last; ++first)
    insert(position, *first);
}

#else /* __STL_MEMBER_TEMPLATES */

template <class T, class Alloc>
void list<T, Alloc>::insert(iterator position, const T* first, const T* last)
{
  for ( ; first != last; ++first)
    insert(position, *first);
```

```
}

template <class T, class Alloc>
void list<T, Alloc>::insert(iterator position,
                        const_iterator first, const_iterator last) {
  for ( ; first != last; ++first)
    insert(position, *first);
}

#endif /* __STL_MEMBER_TEMPLATES */

template <class T, class Alloc>
void list<T, Alloc>::insert(iterator position, size_type n, const T& x) {
  for ( ; n > 0; --n)
    insert(position, x);
}

template <class T, class Alloc>
list<T,Alloc>::iterator list<T, Alloc>::erase(iterator first, iterator last)
{
  while (first != last) erase(first++);
  return last;
}

template <class T, class Alloc>
void list<T, Alloc>::resize(size_type new_size, const T& x)
{
  iterator i = begin();
  size_type len = 0;
  for ( ; i != end() && len < new_size; ++i, ++len)
    ;
  if (len == new_size)
    erase(i, end());
  else                          // i == end()
    insert(end(), new_size - len, x);
}

// 清除所有節點（整個串列）
template <class T, class Alloc>
void list<T, Alloc>::clear()
{
  link_type cur = (link_type) node->next; // begin()
  while (cur != node) {     // 巡訪每一個節點
    link_type tmp = cur;
    cur = (link_type) cur->next;
    destroy_node(tmp);      // 摧毀（解構並釋放）一個節點
  }
  // 恢復 node 原始狀態
  node->next = node;
```

```
  node->prev = node;
}

template <class T, class Alloc>
list<T, Alloc>& list<T, Alloc>::operator=(const list<T, Alloc>& x) {
  if (this != &x) {
    iterator first1 = begin();
    iterator last1 = end();
    const_iterator first2 = x.begin();
    const_iterator last2 = x.end();
    while (first1 != last1 && first2 != last2) *first1++ = *first2++;
    if (first2 == last2)
      erase(first1, last1);
    else
      insert(last1, first2, last2);
  }
  return *this;
}

// 將數值為 value 之所有元素移除
template <class T, class Alloc>
void list<T, Alloc>::remove(const T& value) {
  iterator first = begin();
  iterator last = end();
  while (first != last) {  // 巡訪每一個節點
    iterator next = first;
    ++next;
    if (*first == value) erase(first);    // 找到就移除
    first = next;
  }
}

// 移除數值相同的連續元素
template <class T, class Alloc>
void list<T, Alloc>::unique() {
  iterator first = begin();
  iterator last = end();
  if (first == last) return;
  iterator next = first;
  while (++next != last) {
    if (*first == *next)
      erase(next);
    else
      first = next;
    next = first;
  }
}

// 將 x 合併到 *this 身上。兩個 lists 的內容都必須先經過遞增排序。
```

```
template <class T, class Alloc>
void list<T, Alloc>::merge(list<T, Alloc>& x) {
  iterator first1 = begin();
  iterator last1 = end();
  iterator first2 = x.begin();
  iterator last2 = x.end();

  // 注意：前提是，兩個 lists 都已經過遞增排序，
  while (first1 != last1 && first2 != last2)
    if (*first2 < *first1) {
      iterator next = first2;
      transfer(first1, first2, ++next);
      first2 = next;
    }
    else
      ++first1;
  if (first2 != last2) transfer(last1, first2, last2);
}

// 將 *this 的內容逆向重置
template <class T, class Alloc>
void list<T, Alloc>::reverse() {
  // 以下判斷，如果是空白串列，或僅有一個元素，就不做任何動作。
  // 使用 size() == 0 || size() == 1 來判斷，雖然也可以，但是比較慢。
  if (node->next == node || link_type(node->next)->next == node) return;
  iterator first = begin();
  ++first;
  while (first != end()) {
    iterator old = first;
    ++first;
    transfer(begin(), old, first);
  }
}

// list 不能使用STL 演算法 sort()，必須使用自己的 sort() member function，
// 因為STL演算法sort() 只接受RamdonAccessIterator.
// 本函式採用 quick sort.
template <class T, class Alloc>
void list<T, Alloc>::sort() {
  // 以下判斷，如果是空白串列，或僅有一個元素，就不做任何動作。
  // 使用 size() == 0 || size() == 1 來判斷，雖然也可以，但是比較慢。
  if (node->next == node || link_type(node->next)->next == node) return;

  // 一些新的 lists，做為中介資料存放區
  list<T, Alloc> carry;
  list<T, Alloc> counter[64];
  int fill = 0;
  while (!empty()) {
    carry.splice(carry.begin(), *this, begin());
```

```
    int i = 0;
    while(i < fill && !counter[i].empty()) {
      counter[i].merge(carry);
      carry.swap(counter[i++]);
    }
    carry.swap(counter[i]);
    if (i == fill) ++fill;
  }

  for (int i = 1; i < fill; ++i)
     counter[i].merge(counter[i-1]);
  swap(counter[fill-1]);
}

#ifdef __STL_MEMBER_TEMPLATES

template <class T, class Alloc> template <class Predicate>
void list<T, Alloc>::remove_if(Predicate pred) {
  iterator first = begin();
  iterator last = end();
  while (first != last) {
    iterator next = first;
    ++next;
    if (pred(*first)) erase(first);
    first = next;
  }
}

template <class T, class Alloc> template <class BinaryPredicate>
void list<T, Alloc>::unique(BinaryPredicate binary_pred) {
  iterator first = begin();
  iterator last = end();
  if (first == last) return;
  iterator next = first;
  while (++next != last) {
    if (binary_pred(*first, *next))
      erase(next);
    else
      first = next;
    next = first;
  }
}

template <class T, class Alloc> template <class StrictWeakOrdering>
void list<T, Alloc>::merge(list<T, Alloc>& x, StrictWeakOrdering comp)
{
  iterator first1 = begin();
  iterator last1 = end();
  iterator first2 = x.begin();
```

```
  iterator last2 = x.end();
  while (first1 != last1 && first2 != last2)
    if (comp(*first2, *first1)) {
      iterator next = first2;
      transfer(first1, first2, ++next);
      first2 = next;
    }
    else
      ++first1;
  if (first2 != last2) transfer(last1, first2, last2);
}

template <class T, class Alloc> template <class StrictWeakOrdering>
void list<T, Alloc>::sort(StrictWeakOrdering comp) {
  if (node->next == node || link_type(node->next)->next == node) return;
  list<T, Alloc> carry;
  list<T, Alloc> counter[64];
  int fill = 0;
  while (!empty()) {
    carry.splice(carry.begin(), *this, begin());
    int i = 0;
    while(i < fill && !counter[i].empty()) {
      counter[i].merge(carry, comp);
      carry.swap(counter[i++]);
    }
    carry.swap(counter[i]);
    if (i == fill) ++fill;
  }

  for (int i = 1; i < fill; ++i) counter[i].merge(counter[i-1], comp);
  swap(counter[fill-1]);
}

#endif /* __STL_MEMBER_TEMPLATES */

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

__STL_END_NAMESPACE

#endif /* __SGI_STL_INTERNAL_LIST_H */

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