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

```
/*
 * Copyright (c) 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_SLIST_H
#define __SGI_STL_INTERNAL_SLIST_H


__STL_BEGIN_NAMESPACE

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

// 單向串列的節點基本結構
struct __slist_node_base
{
  __slist_node_base* next;
};

// 全域函式：已知某一節點，安插新節點於其後。
inline __slist_node_base* __slist_make_link(__slist_node_base* prev_node,
                                            __slist_node_base* new_node)
{
  // 令 new 節點的下一節點為prev 節點的下一節點
  new_node->next = prev_node->next;
  prev_node->next = new_node;        // 令 prev 節點的下一節點指向new 節點
  return new_node;
}

// 全域函式：找出某一節點的前一個節點。
inline __slist_node_base* __slist_previous(__slist_node_base* head,
                                           const __slist_node_base* node)
{
```

```
  while (head && head->next != node) // 在單向串列中，只能採用循序搜尋法
    head = head->next;
  return head;
}

// 全域函式：找出某一節點的前一個節點。const 版。
inline const __slist_node_base* __slist_previous(const __slist_node_base* head,
                                                 const __slist_node_base* node)
{
  while (head && head->next != node) // 在單向串列中，只能採用循序搜尋法
    head = head->next;
  return head;
}

// 全域函式：
inline void __slist_splice_after(__slist_node_base* pos,
                                 __slist_node_base* before_first,
                                 __slist_node_base* before_last)
{
  if (pos != before_first && pos != before_last) {
    __slist_node_base* first = before_first->next;
    __slist_node_base* after = pos->next;
    before_first->next = before_last->next;
    pos->next = first;
    before_last->next = after;
  }
}

// 全域函式：
inline __slist_node_base* __slist_reverse(__slist_node_base* node)
{
  __slist_node_base* result = node;
  node = node->next;
  result->next = 0;
  while(node) {
    __slist_node_base* next = node->next;
    node->next = result;
    result = node;
    node = next;
  }
  return result;
}

// 單向串列的節點結構
template <class T>
struct __slist_node : public __slist_node_base
{
  T data;
};
```

```
// 單向串列的迭代器基本結構
struct __slist_iterator_base
{
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;
  typedef forward_iterator_tag iterator_category;  // 注意，是單向

  __slist_node_base* node; // 指向節點基本結構

  __slist_iterator_base(__slist_node_base* x) : node(x) {}

  void incr() { node = node->next; } // 前進一個節點

  bool operator==(const __slist_iterator_base& x) const {
    return node == x.node;
  }
  bool operator!=(const __slist_iterator_base& x) const {
    return node != x.node;
  }
};

// 單向串列的迭代器結構
template <class T, class Ref, class Ptr>
struct __slist_iterator : public __slist_iterator_base
{
  typedef __slist_iterator<T, T&, T*>           iterator;
  typedef __slist_iterator<T, const T&, const T*> const_iterator;
  typedef __slist_iterator<T, Ref, Ptr>         self;

  typedef T value_type;
  typedef Ptr pointer;
  typedef Ref reference;
  typedef __slist_node<T> list_node;

  __slist_iterator(list_node* x) : __slist_iterator_base(x) {}
  // 呼叫 slist<T>::end() 時會造成 __slist_iterator(0)，於是喚起上述函式。
  __slist_iterator() : __slist_iterator_base(0) {}
  __slist_iterator(const iterator& x) : __slist_iterator_base(x.node) {}

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

  self& operator++()
  {
    incr();  // 前進一個節點
    return *this;
```

```
  }
  self operator++(int)
  {
    self tmp = *this;
    incr();  // 前進一個節點
    return tmp;
  }

// 注意，沒有實作 operator--，因為這是一個 forward iterator
// 注意，沒有實作 operator==。於是將使用 __slist_iterator_base::operator==。
// 換句話說兩個 __slist_iterator 的比較，其實就是其底層的兩個 __slist_iterator_base
//   的比較，而也就是比較其內的 __slist_node_base 指標是否相等。
};

#ifndef __STL_CLASS_PARTIAL_SPECIALIZATION

inline ptrdiff_t*
distance_type(const __slist_iterator_base&)
{
  return 0;
}

inline forward_iterator_tag
iterator_category(const __slist_iterator_base&)
{
  return forward_iterator_tag();
}

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

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

// 全域函式：單向串列的大小（元素個數）
inline size_t __slist_size(__slist_node_base* node)
{
  size_t result = 0;
  for ( ; node != 0; node = node->next)
    ++result;     // 累計
  return result;
}

// 單向串列
template <class T, class Alloc = alloc>
class slist
{
```

```
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 size_t size_type;
  typedef ptrdiff_t difference_type;

  typedef __slist_iterator<T, T&, T*>           iterator;
  typedef __slist_iterator<T, const T&, const T*> const_iterator;

private:
  typedef __slist_node<T> list_node;
  typedef __slist_node_base list_node_base;
  typedef __slist_iterator_base iterator_base;
  typedef simple_alloc<list_node, Alloc> list_node_allocator;

  static list_node* create_node(const value_type& x) {
    list_node* node = list_node_allocator::allocate();  // 配置空間
    __STL_TRY {
      construct(&node->data, x);      // 建構元素
      node->next = 0;
    }
    __STL_UNWIND(list_node_allocator::deallocate(node));
    return node;
  }

  static void destroy_node(list_node* node) {
    destroy(&node->data);       // 將元素解構
    list_node_allocator::deallocate(node);    // 釋還空間
  }

  void fill_initialize(size_type n, const value_type& x) {
    head.next = 0;
    __STL_TRY {
      _insert_after_fill(&head, n, x);
    }
    __STL_UNWIND(clear());
  }

#ifdef __STL_MEMBER_TEMPLATES
  template <class InputIterator>
  void range_initialize(InputIterator first, InputIterator last) {
    head.next = 0;
    __STL_TRY {
      _insert_after_range(&head, first, last);
    }
    __STL_UNWIND(clear());
```

```
  }
#else /* __STL_MEMBER_TEMPLATES */
  void range_initialize(const value_type* first, const value_type* last) {
    head.next = 0;
    __STL_TRY {
      _insert_after_range(&head, first, last);
    }
    __STL_UNWIND(clear());
  }
  void range_initialize(const_iterator first, const_iterator last) {
    head.next = 0;
    __STL_TRY {
      _insert_after_range(&head, first, last);
    }
    __STL_UNWIND(clear());
  }
#endif /* __STL_MEMBER_TEMPLATES */

private:
  list_node_base head; // 頭部。注意，它不是指標，是實物。

public:
  slist() { head.next = 0; }

  slist(size_type n, const value_type& x) { fill_initialize(n, x); }
  slist(int n, const value_type& x) { fill_initialize(n, x); }
  slist(long n, const value_type& x) { fill_initialize(n, x); }
  explicit slist(size_type n) { fill_initialize(n, value_type()); }

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

#else /* __STL_MEMBER_TEMPLATES */
  slist(const_iterator first, const_iterator last) {
    range_initialize(first, last);
  }
  slist(const value_type* first, const value_type* last) {
    range_initialize(first, last);
  }
#endif /* __STL_MEMBER_TEMPLATES */

  slist(const slist& L) { range_initialize(L.begin(), L.end()); }

  slist& operator= (const slist& L);

  ~slist() { clear(); }
```

```
public:

  iterator begin() { return iterator((list_node*)head.next); }
  const_iterator begin() const { return const_iterator((list_node*)head.next);}

  iterator end() { return iterator(0); }
  const_iterator end() const { return const_iterator(0); }

  size_type size() const { return __slist_size(head.next); }

  size_type max_size() const { return size_type(-1); }

  bool empty() const { return head.next == 0; }

  // 兩個 slist 互換：只要將 head 交換互指即可。
  void swap(slist& L)
  {
    list_node_base* tmp = head.next;
    head.next = L.head.next;
    L.head.next = tmp;
  }

public:
  friend bool operator== __STL_NULL_TMPL_ARGS(const slist<T, Alloc>& L1,
                                              const slist<T, Alloc>& L2);

public:

  // 取頭部元素
  reference front() { return ((list_node*) head.next)->data; }
  const_reference front() const { return ((list_node*) head.next)->data; }

  // 從頭部安插元素（新元素成為 slist 的第一個元素）
  void push_front(const value_type& x)  {
    __slist_make_link(&head, create_node(x));
  }

  // 注意，沒有 push_back()

  // 從頭部取走元素（刪除之）。修改 head。
  void pop_front() {
    list_node* node = (list_node*) head.next;
    head.next = node->next;
    destroy_node(node);
  }

  iterator previous(const_iterator pos) {
    return iterator((list_node*) __slist_previous(&head, pos.node));
```

```
  }
  const_iterator previous(const_iterator pos) const {
    return const_iterator((list_node*) __slist_previous(&head, pos.node));
  }

private:
  list_node* _insert_after(list_node_base* pos, const value_type& x) {
    return (list_node*) (__slist_make_link(pos, create_node(x)));
  }

  void _insert_after_fill(list_node_base* pos,
                          size_type n, const value_type& x) {
    for (size_type i = 0; i < n; ++i)
      pos = __slist_make_link(pos, create_node(x));
  }

#ifdef __STL_MEMBER_TEMPLATES
  template <class InIter>
  void _insert_after_range(list_node_base* pos, InIter first, InIter last) {
    while (first != last) {
      pos = __slist_make_link(pos, create_node(*first));
      ++first;
    }
  }
#else /* __STL_MEMBER_TEMPLATES */
  void _insert_after_range(list_node_base* pos,
                           const_iterator first, const_iterator last) {
    while (first != last) {
      pos = __slist_make_link(pos, create_node(*first));
      ++first;
    }
  }
  void _insert_after_range(list_node_base* pos,
                           const value_type* first, const value_type* last) {
    while (first != last) {
      pos = __slist_make_link(pos, create_node(*first));
      ++first;
    }
  }
#endif /* __STL_MEMBER_TEMPLATES */

  // 刪除 pos 的下一個元素，並傳回新的下一個元素。
  list_node_base* erase_after(list_node_base* pos) {
    list_node* next = (list_node*) (pos->next);    // 下一個元素
    list_node_base* next_next = next->next;        // 下下一個元素
    pos->next = next_next; // 串接
    destroy_node(next);    // 刪除
    return next_next;      // 傳回
  }
```

```
  list_node_base* erase_after(list_node_base* before_first,
                              list_node_base* last_node) {
    list_node* cur = (list_node*) (before_first->next);
    while (cur != last_node) {
      list_node* tmp = cur;
      cur = (list_node*) cur->next;
      destroy_node(tmp);
    }
    before_first->next = last_node;
    return last_node;
  }


public:

  iterator insert_after(iterator pos, const value_type& x) {
    return iterator(_insert_after(pos.node, x));
  }

  iterator insert_after(iterator pos) {
    return insert_after(pos, value_type());
  }

  void insert_after(iterator pos, size_type n, const value_type& x) {
    _insert_after_fill(pos.node, n, x);
  }
  void insert_after(iterator pos, int n, const value_type& x) {
    _insert_after_fill(pos.node, (size_type) n, x);
  }
  void insert_after(iterator pos, long n, const value_type& x) {
    _insert_after_fill(pos.node, (size_type) n, x);
  }

#ifdef __STL_MEMBER_TEMPLATES
  template <class InIter>
  void insert_after(iterator pos, InIter first, InIter last) {
    _insert_after_range(pos.node, first, last);
  }
#else /* __STL_MEMBER_TEMPLATES */
  void insert_after(iterator pos, const_iterator first, const_iterator last) {
    _insert_after_range(pos.node, first, last);
  }
  void insert_after(iterator pos,
                    const value_type* first, const value_type* last) {
    _insert_after_range(pos.node, first, last);
  }
#endif /* __STL_MEMBER_TEMPLATES */
```

```
  iterator insert(iterator pos, const value_type& x) {
    return iterator(_insert_after(__slist_previous(&head, pos.node), x));
  }

  iterator insert(iterator pos) {
    return iterator(_insert_after(__slist_previous(&head, pos.node),
                                  value_type()));
  }

  void insert(iterator pos, size_type n, const value_type& x) {
    _insert_after_fill(__slist_previous(&head, pos.node), n, x);
  }
  void insert(iterator pos, int n, const value_type& x) {
    _insert_after_fill(__slist_previous(&head, pos.node), (size_type) n, x);
  }
  void insert(iterator pos, long n, const value_type& x) {
    _insert_after_fill(__slist_previous(&head, pos.node), (size_type) n, x);
  }

#ifdef __STL_MEMBER_TEMPLATES
  template <class InIter>
  void insert(iterator pos, InIter first, InIter last) {
    _insert_after_range(__slist_previous(&head, pos.node), first, last);
  }
#else /* __STL_MEMBER_TEMPLATES */
  void insert(iterator pos, const_iterator first, const_iterator last) {
    _insert_after_range(__slist_previous(&head, pos.node), first, last);
  }
  void insert(iterator pos, const value_type* first, const value_type* last) {
    _insert_after_range(__slist_previous(&head, pos.node), first, last);
  }
#endif /* __STL_MEMBER_TEMPLATES */


public:
  iterator erase_after(iterator pos) {
    return iterator((list_node*)erase_after(pos.node));
  }
  iterator erase_after(iterator before_first, iterator last) {
    return iterator((list_node*)erase_after(before_first.node, last.node));
  }

  iterator erase(iterator pos) {
    return (list_node*) erase_after(__slist_previous(&head, pos.node));
  }
  iterator erase(iterator first, iterator last) {
    return (list_node*) erase_after(__slist_previous(&head, first.node),
                                    last.node);
  }
```

```
  void resize(size_type new_size, const T& x);
  void resize(size_type new_size) { resize(new_size, T()); }
  void clear() { erase_after(&head, 0); }

public:
  // Moves the range [before_first + 1, before_last + 1) to *this,
  //  inserting it immediately after pos.  This is constant time.
  void splice_after(iterator pos,
                    iterator before_first, iterator before_last)
  {
    if (before_first != before_last)
      __slist_splice_after(pos.node, before_first.node, before_last.node);
  }

  // Moves the element that follows prev to *this, inserting it immediately
  //  after pos.  This is constant time.
  void splice_after(iterator pos, iterator prev)
  {
    __slist_splice_after(pos.node, prev.node, prev.node->next);
  }


  // Linear in distance(begin(), pos), and linear in L.size().
  void splice(iterator pos, slist& L) {
    if (L.head.next)
      __slist_splice_after(__slist_previous(&head, pos.node),
                           &L.head,
                           __slist_previous(&L.head, 0));
  }

  // Linear in distance(begin(), pos), and in distance(L.begin(), i).
  void splice(iterator pos, slist& L, iterator i) {
    __slist_splice_after(__slist_previous(&head, pos.node),
                         __slist_previous(&L.head, i.node),
                         i.node);
  }

  // Linear in distance(begin(), pos), in distance(L.begin(), first),
  // and in distance(first, last).
  void splice(iterator pos, slist& L, iterator first, iterator last)
  {
    if (first != last)
      __slist_splice_after(__slist_previous(&head, pos.node),
                           __slist_previous(&L.head, first.node),
                           __slist_previous(first.node, last.node));
  }

public:
```

```
  void reverse() { if (head.next) head.next = __slist_reverse(head.next); }

  void remove(const T& val);
  void unique();
  void merge(slist& L);
  void sort();

#ifdef __STL_MEMBER_TEMPLATES
  template <class Predicate> void remove_if(Predicate pred);
  template <class BinaryPredicate> void unique(BinaryPredicate pred);
  template <class StrictWeakOrdering> void merge(slist&, StrictWeakOrdering);
  template <class StrictWeakOrdering> void sort(StrictWeakOrdering comp);
#endif /* __STL_MEMBER_TEMPLATES */
};

template <class T, class Alloc>
slist<T, Alloc>& slist<T,Alloc>::operator=(const slist<T, Alloc>& L)
{
  if (&L != this) {
    list_node_base* p1 = &head;
    list_node* n1 = (list_node*) head.next;
    const list_node* n2 = (const list_node*) L.head.next;
    while (n1 && n2) {
      n1->data = n2->data;
      p1 = n1;
      n1 = (list_node*) n1->next;
      n2 = (const list_node*) n2->next;
    }
    if (n2 == 0)
      erase_after(p1, 0);
    else
      _insert_after_range(p1,
                          const_iterator((list_node*)n2), const_iterator(0));
  }
  return *this;
}

template <class T, class Alloc>
bool operator==(const slist<T, Alloc>& L1, const slist<T, Alloc>& L2)
{
  typedef typename slist<T,Alloc>::list_node list_node;
  list_node* n1 = (list_node*) L1.head.next;
  list_node* n2 = (list_node*) L2.head.next;
  while (n1 && n2 && n1->data == n2->data) {
    n1 = (list_node*) n1->next;
    n2 = (list_node*) n2->next;
  }
  return n1 == 0 && n2 == 0;
}
```

```
template <class T, class Alloc>
inline bool operator<(const slist<T, Alloc>& L1, const slist<T, Alloc>& L2)
{
  return lexicographical_compare(L1.begin(), L1.end(), L2.begin(), L2.end());
}

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

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

#endif /* __STL_FUNCTION_TMPL_PARTIAL_ORDER */


template <class T, class Alloc>
void slist<T, Alloc>::resize(size_type len, const T& x)
{
  list_node_base* cur = &head;
  while (cur->next != 0 && len > 0) {
    --len;
    cur = cur->next;
  }
  if (cur->next)
    erase_after(cur, 0);
  else
    _insert_after_fill(cur, len, x);
}

template <class T, class Alloc>
void slist<T,Alloc>::remove(const T& val)
{
  list_node_base* cur = &head;
  while (cur && cur->next) {
    if (((list_node*) cur->next)->data == val)
      erase_after(cur);
    else
      cur = cur->next;
  }
}

template <class T, class Alloc>
void slist<T,Alloc>::unique()
{
  list_node_base* cur = head.next;
  if (cur) {
    while (cur->next) {
```

```
      if (((list_node*)cur)->data == ((list_node*)(cur->next))->data)
        erase_after(cur);
      else
        cur = cur->next;
    }
  }
}

template <class T, class Alloc>
void slist<T,Alloc>::merge(slist<T,Alloc>& L)
{
  list_node_base* n1 = &head;
  while (n1->next && L.head.next) {
    if (((list_node*) L.head.next)->data < ((list_node*) n1->next)->data)
      __slist_splice_after(n1, &L.head, L.head.next);
    n1 = n1->next;
  }
  if (L.head.next) {
    n1->next = L.head.next;
    L.head.next = 0;
  }
}

template <class T, class Alloc>
void slist<T,Alloc>::sort()
{
  if (head.next && head.next->next) {
    slist carry;
    slist counter[64];
    int fill = 0;
    while (!empty()) {
      __slist_splice_after(&carry.head, &head, head.next);
      int i = 0;
      while (i < fill && !counter[i].empty()) {
        counter[i].merge(carry);
        carry.swap(counter[i]);
        ++i;
      }
      carry.swap(counter[i]);
      if (i == fill)
        ++fill;
    }

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

```
#ifdef __STL_MEMBER_TEMPLATES

template <class T, class Alloc>
template <class Predicate> void slist<T,Alloc>::remove_if(Predicate pred)
{
  list_node_base* cur = &head;
  while (cur->next) {
    if (pred(((list_node*) cur->next)->data))
      erase_after(cur);
    else
      cur = cur->next;
  }
}

template <class T, class Alloc> template <class BinaryPredicate>
void slist<T,Alloc>::unique(BinaryPredicate pred)
{
  list_node* cur = (list_node*) head.next;
  if (cur) {
    while (cur->next) {
      if (pred(((list_node*)cur)->data, ((list_node*)(cur->next))->data))
        erase_after(cur);
      else
        cur = (list_node*) cur->next;
    }
  }
}

template <class T, class Alloc> template <class StrictWeakOrdering>
void slist<T,Alloc>::merge(slist<T,Alloc>& L, StrictWeakOrdering comp)
{
  list_node_base* n1 = &head;
  while (n1->next && L.head.next) {
    if (comp(((list_node*) L.head.next)->data,
             ((list_node*) n1->next)->data))
      __slist_splice_after(n1, &L.head, L.head.next);
    n1 = n1->next;
  }
  if (L.head.next) {
    n1->next = L.head.next;
    L.head.next = 0;
  }
}

template <class T, class Alloc> template <class StrictWeakOrdering>
void slist<T,Alloc>::sort(StrictWeakOrdering comp)
{
  if (head.next && head.next->next) {
    slist carry;
```

```
    slist counter[64];
    int fill = 0;
    while (!empty()) {
      __slist_splice_after(&carry.head, &head, head.next);
      int i = 0;
      while (i < fill && !counter[i].empty()) {
        counter[i].merge(carry, comp);
        carry.swap(counter[i]);
        ++i;
      }
      carry.swap(counter[i]);
      if (i == fill)
        ++fill;
    }

    for (int i = 1; i < fill; ++i)
      counter[i].merge(counter[i-1], comp);
    this->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_SLIST_H */

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