G++ 2.91.57，cygnus\cygwin-b20\include\g++**stl_algobase.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
 * 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_ALGOBASE_H
#define __SGI_STL_INTERNAL_ALGOBASE_H

#ifndef __STL_CONFIG_H
#include <stl_config.h>
#endif
#ifndef __SGI_STL_INTERNAL_RELOPS
#include <stl_relops.h>
#endif
#ifndef __SGI_STL_INTERNAL_PAIR_H
#include <stl_pair.h>
#endif
#ifndef __TYPE_TRAITS_H_
#include <type_traits.h>
#endif

#include <string.h>
```

```
#include <limits.h>
#include <stdlib.h>
#include <stddef.h>
#include <new.h>
#include <iostream.h>

#ifndef __SGI_STL_INTERNAL_ITERATOR_H
#include <stl_iterator.h>
#endif

__STL_BEGIN_NAMESPACE

template <class ForwardIterator1, class ForwardIterator2, class T>
inline void __iter_swap(ForwardIterator1 a, ForwardIterator2 b, T*) {
  T tmp = *a;
  *a = *b;
  *b = tmp;
}

template <class ForwardIterator1, class ForwardIterator2>
inline void iter_swap(ForwardIterator1 a, ForwardIterator2 b) {
  // iter_swap() 是「有必要運用迭代器之 value type」的一個好例子。
  // 是的，它必須知道迭代器的 value type，才能夠據此宣告一個物件，用來
  // 暫時放置迭代器所指的物件。
  __iter_swap(a, b, value_type(a)); // 注意第三參數的型別!
  /*
  // 以下定義於 <stl_iterator.h>
  template <class Iterator>
  inline typename iterator_traits<Iterator>::value_type*
  value_type(const Iterator&) {
    return static_cast<typename iterator_traits<Iterator>::value_type*>(0);
  }
  */

  // 侯捷認為（並予實證），不需像上行那樣轉呼叫，可改用以下寫法：
  // typename iterator_traits<ForwardIterator1>::value_type tmp = *a;
  // *a = *b;
  // *b = tmp;
}

template <class T>
inline void swap(T& a, T& b) {
  T tmp = a;
  a = b;
  b = tmp;
}

#ifndef __BORLANDC__
```

```
#undef min
#undef max

template <class T>
inline const T& min(const T& a, const T& b) {
  return b < a ? b : a;
}

template <class T>
inline const T& max(const T& a, const T& b) {
  return  a < b ? b : a;
}

#endif /* __BORLANDC__ */

template <class T, class Compare>
inline const T& min(const T& a, const T& b, Compare comp) {
  return comp(b, a) ? b : a;    // 由 comp 決定「大小比較」標準
}

template <class T, class Compare>
inline const T& max(const T& a, const T& b, Compare comp) {
  return comp(a, b) ? b : a;    // 由 comp 決定「大小比較」標準
}

template <class InputIterator, class OutputIterator>
inline OutputIterator __copy(InputIterator first, InputIterator last,
                         OutputIterator result, input_iterator_tag)
{
  for ( ; first != last; ++result, ++first)
    *result = *first;
  return result;
}

template <class RandomAccessIterator, class OutputIterator, class Distance>
inline OutputIterator
__copy_d(RandomAccessIterator first, RandomAccessIterator last,
         OutputIterator result, Distance*)
{
  for (Distance n = last - first; n > 0; --n, ++result, ++first)
    *result = *first;
  return result;
}

template <class RandomAccessIterator, class OutputIterator>
inline OutputIterator
__copy(RandomAccessIterator first, RandomAccessIterator last,
       OutputIterator result, random_access_iterator_tag)
{
```

```
  return __copy_d(first, last, result, distance_type(first));
}

template <class InputIterator, class OutputIterator>
struct __copy_dispatch
{
  OutputIterator operator()(InputIterator first, InputIterator last,
                            OutputIterator result) {
    return __copy(first, last, result, iterator_category(first));
  }
};

#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION

template <class T>
inline T* __copy_t(const T* first, const T* last, T* result, __true_type) {
  memmove(result, first, sizeof(T) * (last - first));
  return result + (last - first);
}

template <class T>
inline T* __copy_t(const T* first, const T* last, T* result, __false_type) {
  return __copy_d(first, last, result, (ptrdiff_t*) 0);
}

template <class T>
struct __copy_dispatch<T*, T*>
{
  T* operator()(T* first, T* last, T* result) {
    typedef typename __type_traits<T>::has_trivial_assignment_operator t;
    return __copy_t(first, last, result, t());
  }
};

template <class T>
struct __copy_dispatch<const T*, T*>
{
  T* operator()(const T* first, const T* last, T* result) {
    typedef typename __type_traits<T>::has_trivial_assignment_operator t;
    return __copy_t(first, last, result, t());
  }
};

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

// copy 函式運用了 function overloading, type traits, partial
// specialization, 無所不用其極地改善效率。
template <class InputIterator, class OutputIterator>
inline OutputIterator copy(InputIterator first, InputIterator last,
```

```
                      OutputIterator result)
{
  return __copy_dispatch<InputIterator,OutputIterator>()(first, last, result);
}

inline char* copy(const char* first, const char* last, char* result) {
  memmove(result, first, last - first);
  return result + (last - first);
}

inline wchar_t* copy(const wchar_t* first, const wchar_t* last,
                     wchar_t* result) {
  memmove(result, first, sizeof(wchar_t) * (last - first));
  return result + (last - first);
}

template <class BidirectionalIterator1, class BidirectionalIterator2>
inline BidirectionalIterator2 __copy_backward(BidirectionalIterator1 first,
                                              BidirectionalIterator1 last,
                                              BidirectionalIterator2 result) {
  while (first != last) *--result = *--last;
  return result;
}


template <class BidirectionalIterator1, class BidirectionalIterator2>
struct __copy_backward_dispatch
{
  BidirectionalIterator2 operator()(BidirectionalIterator1 first,
                                    BidirectionalIterator1 last,
                                    BidirectionalIterator2 result) {
    return __copy_backward(first, last, result);
  }
};

#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION

template <class T>
inline T* __copy_backward_t(const T* first, const T* last, T* result,
                            __true_type) {
  const ptrdiff_t N = last - first;
  memmove(result - N, first, sizeof(T) * N);
  return result - N;
}

template <class T>
inline T* __copy_backward_t(const T* first, const T* last, T* result,
                            __false_type) {
  return __copy_backward(first, last, result);
```

```
}

template <class T>
struct __copy_backward_dispatch<T*, T*>
{
  T* operator()(T* first, T* last, T* result) {
    typedef typename __type_traits<T>::has_trivial_assignment_operator t;
    return __copy_backward_t(first, last, result, t());
  }
};

template <class T>
struct __copy_backward_dispatch<const T*, T*>
{
  T* operator()(const T* first, const T* last, T* result) {
    typedef typename __type_traits<T>::has_trivial_assignment_operator t;
    return __copy_backward_t(first, last, result, t());
  }
};

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

template <class BidirectionalIterator1, class BidirectionalIterator2>
inline BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,
                                            BidirectionalIterator1 last,
                                            BidirectionalIterator2 result) {
  return __copy_backward_dispatch<BidirectionalIterator1,
                                  BidirectionalIterator2>()(first, last,
                                                            result);
}

template <class InputIterator, class Size, class OutputIterator>
pair<InputIterator, OutputIterator> __copy_n(InputIterator first, Size count,
                                             OutputIterator result,
                                             input_iterator_tag) {
  for ( ; count > 0; --count, ++first, ++result)
    *result = *first;
  return pair<InputIterator, OutputIterator>(first, result);
}

template <class RandomAccessIterator, class Size, class OutputIterator>
inline pair<RandomAccessIterator, OutputIterator>
__copy_n(RandomAccessIterator first, Size count,
         OutputIterator result,
         random_access_iterator_tag) {
  RandomAccessIterator last = first + count;
  return pair<RandomAccessIterator, OutputIterator>(last,
                                                    copy(first, last, result));
}
```

```
// 以下為 SGI STL 專屬，從 first 開始複製 count 個元素到 result 以後的空間。
template <class InputIterator, class Size, class OutputIterator>
inline pair<InputIterator, OutputIterator>
copy_n(InputIterator first, Size count,
       OutputIterator result) {
  return __copy_n(first, count, result, iterator_category(first));
}

template <class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value) {
  for ( ; first != last; ++first)    // 迭代走過整個範圍
    *first = value;
}

template <class OutputIterator, class Size, class T>
OutputIterator fill_n(OutputIterator first, Size n, const T& value) {
  for ( ; n > 0; --n, ++first)       // 經過n個元素
    *first = value;    // 注意，assignment 是覆寫（overwrite）而不是安插（insert）
  return first;
}

template <class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1,
                              InputIterator1 last1,
                              InputIterator2 first2) {
  // 以下，如果序列一走完，就結束。
  // 以下，如果序列一和序列二的對應元素相等，就結束。
  // 顯然，序列一的元素個數必須多過序列二的元素個數，否則結果無可預期。
  while (first1 != last1 && *first1 == *first2) {
    ++first1;
    ++first2;
  }
  return pair<InputIterator1, InputIterator2>(first1, first2);
}

template <class InputIterator1, class InputIterator2, class BinaryPredicate>
pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1,
                              InputIterator1 last1,
                              InputIterator2 first2,
                              BinaryPredicate binary_pred) {
  while (first1 != last1 && binary_pred(*first1, *first2)) {
    ++first1;
    ++first2;
  }
  return pair<InputIterator1, InputIterator2>(first1, first2);
}

template <class InputIterator1, class InputIterator2>
```

```
inline bool equal(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2) {
  // 以下，將序列一走過一遍。序列二亦步亦趨。
  // 如果序列一的元素個數多過序列二的元素個數，就糟糕了。
  for ( ; first1 != last1; ++first1, ++first2)
    if (*first1 != *first2)     // 只要對應元素不相等，
      return false;             // 就結束並傳回 false。
  return true;                  // 至此，全部相等，傳回true。
}

template <class InputIterator1, class InputIterator2, class BinaryPredicate>
inline bool equal(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, BinaryPredicate binary_pred) {
  for ( ; first1 != last1; ++first1, ++first2)
    if (!binary_pred(*first1, *first2))
      return false;
  return true;
}

template <class InputIterator1, class InputIterator2>
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                 InputIterator2 first2, InputIterator2 last2) {
  // 以下，任何一個序列到達尾端，就結束。否則兩序列就相應元素一一進行比對。
  for ( ; first1 != last1 && first2 != last2; ++first1, ++first2) {
    if (*first1 < *first2) // 第一序列元素值小於第二序列的相應元素值
      return true;
    if (*first2 < *first1) // 第二序列元素值小於第一序列的相應元素值
      return false;
    // 如果不符合以上兩條件，表示兩值相等，那就進行下一組相應元素值的比對。
  }
  // 進行到這裡，如果第一序列到達尾端而第二序列尚有餘額，那麼第一序列小於第二序列。
  return first1 == last1 && first2 != last2;
}

template <class InputIterator1, class InputIterator2, class Compare>
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                 InputIterator2 first2, InputIterator2 last2,
                 Compare comp) {
  for ( ; first1 != last1 && first2 != last2; ++first1, ++first2) {
    if (comp(*first1, *first2))
      return true;
    if (comp(*first2, *first1))
      return false;
  }
  return first1 == last1 && first2 != last2;
}

inline bool
lexicographical_compare(const unsigned char* first1,
```

```
                    const unsigned char* last1,
                    const unsigned char* first2,
                    const unsigned char* last2)
{
  const size_t len1 = last1 - first1;    // 第一序列長度
  const size_t len2 = last2 - first2;    // 第二序列長度
  // 先比較相同長度的一截。memcmp() 速度極快。
  const int result = memcmp(first1, first2, min(len1, len2));
  // 如果不相上下，則長度較長者被視為比較大。
  return result != 0 ? result < 0 : len1 < len2;
}

inline bool lexicographical_compare(const char* first1, const char* last1,
                                    const char* first2, const char* last2)
{
#if CHAR_MAX == SCHAR_MAX
  // 轉型為 const signed char*
  return lexicographical_compare((const signed char*) first1,
                                 (const signed char*) last1,
                                 (const signed char*) first2,
                                 (const signed char*) last2);
#else
  // 轉型為 const unsigned char*
  return lexicographical_compare((const unsigned char*) first1,
                                 (const unsigned char*) last1,
                                 (const unsigned char*) first2,
                                 (const unsigned char*) last2);
#endif
}

template <class InputIterator1, class InputIterator2>
int lexicographical_compare_3way(InputIterator1 first1, InputIterator1 last1,
                                 InputIterator2 first2, InputIterator2 last2)
{
  while (first1 != last1 && first2 != last2) {
    if (*first1 < *first2) return -1;
    if (*first2 < *first1) return 1;
    ++first1; ++first2;
  }
  if (first2 == last2) {
    return !(first1 == last1);
  } else {
    return -1;
  }
}

inline int
lexicographical_compare_3way(const unsigned char* first1,
                             const unsigned char* last1,
```

```
                        const unsigned char* first2,
                        const unsigned char* last2)
{
  const ptrdiff_t len1 = last1 - first1;
  const ptrdiff_t len2 = last2 - first2;
  const int result = memcmp(first1, first2, min(len1, len2));
  return result != 0 ? result : (len1 == len2 ? 0 : (len1 < len2 ? -1 : 1));
}

inline int lexicographical_compare_3way(const char* first1, const char* last1,
                                        const char* first2, const char* last2)
{
#if CHAR_MAX == SCHAR_MAX
  return lexicographical_compare_3way(
                 (const signed char*) first1,
                           (const signed char*) last1,
                           (const signed char*) first2,
                           (const signed char*) last2);
#else
  return lexicographical_compare_3way((const unsigned char*) first1,
                                      (const unsigned char*) last1,
                                      (const unsigned char*) first2,
                                      (const unsigned char*) last2);
#endif
}

__STL_END_NAMESPACE

#endif /* __SGI_STL_INTERNAL_ALGOBASE_H */

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