G++ 2.91.57，cygnus\cygwin-b20\include\g++**stl_iterator.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_ITERATOR_H
#define __SGI_STL_INTERNAL_ITERATOR_H

__STL_BEGIN_NAMESPACE

// 五種迭代器類型
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : public input_iterator_tag {};
struct bidirectional_iterator_tag : public forward_iterator_tag {};
struct random_access_iterator_tag : public bidirectional_iterator_tag {};

template <class T, class Distance> struct input_iterator {
  typedef input_iterator_tag iterator_category;
  typedef T                  value_type;
  typedef Distance           difference_type;
  typedef T*                 pointer;
  typedef T&                 reference;
```

```
};

struct output_iterator {
  typedef output_iterator_tag iterator_category;
  typedef void              value_type;
  typedef void              difference_type;
  typedef void              pointer;
  typedef void              reference;
};

template <class T, class Distance> struct forward_iterator {
  typedef forward_iterator_tag iterator_category;
  typedef T                 value_type;
  typedef Distance           difference_type;
  typedef T*                pointer;
  typedef T&                reference;
};


template <class T, class Distance> struct bidirectional_iterator {
  typedef bidirectional_iterator_tag iterator_category;
  typedef T                       value_type;
  typedef Distance                 difference_type;
  typedef T*                      pointer;
  typedef T&                      reference;
};

template <class T, class Distance> struct random_access_iterator {
  typedef random_access_iterator_tag iterator_category;
  typedef T                       value_type;
  typedef Distance                 difference_type;
  typedef T*                      pointer;
  typedef T&                      reference;
};

#ifdef __STL_USE_NAMESPACES
// 為避免寫碼時掛一漏萬，自行開發的迭代器最好繼承自下面這個 std::iterator
template <class Category, class T, class Distance = ptrdiff_t,
        class Pointer = T*, class Reference = T&>
struct iterator {
  typedef Category  iterator_category;
  typedef T        value_type;
  typedef Distance  difference_type;
  typedef Pointer   pointer;
  typedef Reference reference;
};
#endif /* __STL_USE_NAMESPACES */

#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION
```

```
// 以下是在支援 partial specialization 的編譯器上的實作方法

template <class Iterator>
struct iterator_traits {
  typedef typename Iterator::iterator_category iterator_category;
  typedef typename Iterator::value_type       value_type;
  typedef typename Iterator::difference_type   difference_type;
  typedef typename Iterator::pointer          pointer;
  typedef typename Iterator::reference        reference;
};

// 針對原生指標（native pointer）而設計的 traits 偏特化版。
template <class T>
struct iterator_traits<T*> {
  typedef random_access_iterator_tag iterator_category;
  typedef T                         value_type;
  typedef ptrdiff_t                  difference_type;
  typedef T*                        pointer;
  typedef T&                        reference;
};

// 針對原生之 pointer-to-const 而設計的 traits 偏特化版。
template <class T>
struct iterator_traits<const T*> {
  typedef random_access_iterator_tag iterator_category;
  typedef T                         value_type;
  typedef ptrdiff_t                  difference_type;
  typedef const T*                   pointer;
  typedef const T&                   reference;
};

// 這個函式可以很方便地決定某個迭代器的類型（category）
template <class Iterator>
inline typename iterator_traits<Iterator>::iterator_category
iterator_category(const Iterator&) {
  typedef typename iterator_traits<Iterator>::iterator_category category;
  return category();
}

// 這個函式可以很方便地決定某個迭代器的 distance type
template <class Iterator>
inline typename iterator_traits<Iterator>::difference_type*
distance_type(const Iterator&) {
  return static_cast<typename iterator_traits<Iterator>::difference_type*>(0);
}

// 這個函式可以很方便地決定某個迭代器的 value type
template <class Iterator>
inline typename iterator_traits<Iterator>::value_type*
```

```
value_type(const Iterator&) {
  return static_cast<typename iterator_traits<Iterator>::value_type*>(0);
}

#else /* __STL_CLASS_PARTIAL_SPECIALIZATION */
// 以下是在未支援 partial specialization 的編譯器上的實作方法

template <class T, class Distance>
inline input_iterator_tag
iterator_category(const input_iterator<T, Distance>&) {
  return input_iterator_tag();
}

inline output_iterator_tag iterator_category(const output_iterator&) {
  return output_iterator_tag();
}

template <class T, class Distance>
inline forward_iterator_tag
iterator_category(const forward_iterator<T, Distance>&) {
  return forward_iterator_tag();
}

template <class T, class Distance>
inline bidirectional_iterator_tag
iterator_category(const bidirectional_iterator<T, Distance>&) {
  return bidirectional_iterator_tag();
}

template <class T, class Distance>
inline random_access_iterator_tag
iterator_category(const random_access_iterator<T, Distance>&) {
  return random_access_iterator_tag();
}

template <class T>
inline random_access_iterator_tag iterator_category(const T*) {
  return random_access_iterator_tag();
}

template <class T, class Distance>
inline T* value_type(const input_iterator<T, Distance>&) {
  return (T*)(0);
}

template <class T, class Distance>
inline T* value_type(const forward_iterator<T, Distance>&) {
  return (T*)(0);
}
```

```
template <class T, class Distance>
inline T* value_type(const bidirectional_iterator<T, Distance>&) {
  return (T*)(0);
}

template <class T, class Distance>
inline T* value_type(const random_access_iterator<T, Distance>&) {
  return (T*)(0);
}

template <class T>
inline T* value_type(const T*) { return (T*)(0); }

template <class T, class Distance>
inline Distance* distance_type(const input_iterator<T, Distance>&) {
  return (Distance*)(0);
}

template <class T, class Distance>
inline Distance* distance_type(const forward_iterator<T, Distance>&) {
  return (Distance*)(0);
}

template <class T, class Distance>
inline Distance*
distance_type(const bidirectional_iterator<T, Distance>&) {
  return (Distance*)(0);
}

template <class T, class Distance>
inline Distance*
distance_type(const random_access_iterator<T, Distance>&) {
  return (Distance*)(0);
}

template <class T>
inline ptrdiff_t* distance_type(const T*) { return (ptrdiff_t*)(0); }

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */


// 以下是整組 distance 函式
template <class InputIterator, class Distance>
inline void __distance(InputIterator first, InputIterator last, Distance& n,
                       input_iterator_tag) {
  while (first != last) { ++first; ++n; }
}
```

```
template <class RandomAccessIterator, class Distance>
inline void __distance(RandomAccessIterator first, RandomAccessIterator last,
                       Distance& n, random_access_iterator_tag) {
  n += last - first;
}

template <class InputIterator, class Distance>
inline void distance(InputIterator first, InputIterator last, Distance& n)
{
  __distance(first, last, n, iterator_category(first));
}

#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION
// 以下是在支援 partial specialization 的編譯器上的實作方法

template <class InputIterator>
inline iterator_traits<InputIterator>::difference_type
__distance(InputIterator first, InputIterator last, input_iterator_tag) {
  iterator_traits<InputIterator>::difference_type n = 0;
  while (first != last) {
    ++first; ++n;
  }
  return n;
}

template <class RandomAccessIterator>
inline iterator_traits<RandomAccessIterator>::difference_type
__distance(RandomAccessIterator first, RandomAccessIterator last,
           random_access_iterator_tag) {
  return last - first;
}

template <class InputIterator>
inline iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last) {
  typedef typename iterator_traits<InputIterator>::iterator_category category;
  return __distance(first, last, category());
}

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */


// 以下是整組 advance 函式
template <class InputIterator, class Distance>
inline void __advance(InputIterator& i, Distance n, input_iterator_tag) {
  while (n--) ++i;
}

#if defined(__sgi) && !defined(__GNUC__) && (_MIPS_SIM != _MIPS_SIM_ABI32)
```

```
#pragma set woff 1183
#endif

template <class BidirectionalIterator, class Distance>
inline void __advance(BidirectionalIterator& i, Distance n,
                      bidirectional_iterator_tag) {
  if (n >= 0)
    while (n--) ++i;
  else
    while (n++) --i;
}

#if defined(__sgi) && !defined(__GNUC__) && (_MIPS_SIM != _MIPS_SIM_ABI32)
#pragma reset woff 1183
#endif

template <class RandomAccessIterator, class Distance>
inline void __advance(RandomAccessIterator& i, Distance n,
                      random_access_iterator_tag) {
  i += n;
}

template <class InputIterator, class Distance>
inline void advance(InputIterator& i, Distance n) {
  __advance(i, n, iterator_category(i));
}

// 這是一個迭代器配接器（iterator adapter），用來將某個迭代器的賦值（assign）
// 動作修改為安插（insert）動作 — 從容器的尾端安插進去。
template <class Container>
class back_insert_iterator {
protected:
  Container* container;
public:
  typedef output_iterator_tag   iterator_category;
  typedef void                  value_type;
  typedef void                  difference_type;
  typedef void                  pointer;
  typedef void                  reference;

  // 下面這個 ctor 使 back_insert_iterator 與容器x繫結起來。
  explicit back_insert_iterator(Container& x) : container(&x) {}
  back_insert_iterator<Container>&
  operator=(const typename Container::value_type& value) {
    container->push_back(value);
    return *this;
  }
  // 以下三個運算子對 back_insert_iterator 不起作用（關閉功能）
  // 三個運算子傳回的都是 back_insert_iterator 自己。
```

```
  back_insert_iterator<Container>& operator*() { return *this; }
  back_insert_iterator<Container>& operator++() { return *this; }
  back_insert_iterator<Container>& operator++(int) { return *this; }
};

#ifndef __STL_CLASS_PARTIAL_SPECIALIZATION

template <class Container>
inline output_iterator_tag
iterator_category(const back_insert_iterator<Container>&)
{
  return output_iterator_tag();
}

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

// 這是一個輔助函式，幫助我們方便使用 back_insert_iterator。
template <class Container>
inline back_insert_iterator<Container> back_inserter(Container& x) {
  return back_insert_iterator<Container>(x);
}

// 這是一個迭代器配接器（iterator adapter），用來將某個迭代器的賦值（assign）
// 動作修改為安插（insert）動作 — 從容器的頭端安插進去。
template <class Container>
class front_insert_iterator {
protected:
  Container* container;
public:
  typedef output_iterator_tag   iterator_category;
  typedef void                  value_type;
  typedef void                  difference_type;
  typedef void                  pointer;
  typedef void                  reference;

  // 下面這個 ctor 使 front_insert_iterator 與容器x繫結起來。
  explicit front_insert_iterator(Container& x) : container(&x) {}
  front_insert_iterator<Container>&
  operator=(const typename Container::value_type& value) {
    container->push_front(value);
    return *this;
  }
  // 以下三個運算子對 front_insert_iterator 不起作用（關閉功能）
  // 三個運算子傳回的都是 front_insert_iterator 自己。
  front_insert_iterator<Container>& operator*() { return *this; }
  front_insert_iterator<Container>& operator++() { return *this; }
  front_insert_iterator<Container>& operator++(int) { return *this; }
};
```

```
#ifndef __STL_CLASS_PARTIAL_SPECIALIZATION

template <class Container>
inline output_iterator_tag
iterator_category(const front_insert_iterator<Container>&)
{
  return output_iterator_tag();
}

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

// 這是一個輔助函式，幫助我們方便使用 front_insert_iterator。
template <class Container>
inline front_insert_iterator<Container> front_inserter(Container& x) {
  return front_insert_iterator<Container>(x);
}

// 這是一個迭代器配接器（iterator adapter），用來將某個迭代器的賦值（assign）
// 動作修改為安插（insert）動作，從指定的位置安插進去，並將迭代器前進一個位置
// — 如此便可單純地連續執行「表面上是賦值（覆寫）而實際上是安插」的動作。
template <class Container>
class insert_iterator {
protected:
  Container* container;
  typename Container::iterator iter;
public:
  typedef output_iterator_tag   iterator_category;
  typedef void                  value_type;
  typedef void                  difference_type;
  typedef void                  pointer;
  typedef void                  reference;

  // 下面這個 ctor 使 insert_iterator 與容器x和迭代器 i繫結起來。
  insert_iterator(Container& x, typename Container::iterator i)
    : container(&x), iter(i) {}
  insert_iterator<Container>&
  operator=(const typename Container::value_type& value) {
    iter = container->insert(iter, value);
    ++iter;  // 注意這個，使 insert iterator 永遠隨其標的物貼身移動
    return *this;
  }
  // 以下三個運算子對 insert_iterator 不起作用（關閉功能）
  // 三個運算子傳回的都是 insert_iterator 自己。
  insert_iterator<Container>& operator*() { return *this; }
  insert_iterator<Container>& operator++() { return *this; }
  insert_iterator<Container>& operator++(int) { return *this; }
};

#ifndef __STL_CLASS_PARTIAL_SPECIALIZATION
```

```
template <class Container>
inline output_iterator_tag
iterator_category(const insert_iterator<Container>&)
{
  return output_iterator_tag();
}

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

// 這是一個輔助函式，幫助我們方便使用 insert_iterator。
template <class Container, class Iterator>
inline insert_iterator<Container> inserter(Container& x, Iterator i) {
  typedef typename Container::iterator iter;
  return insert_iterator<Container>(x, iter(i));
}

// 這是一個迭代器配接器（iterator adapter），用來將某個雙向迭代器逆反前進方向，
// 使前進為後退，後退為前進。
#ifndef __STL_LIMITED_DEFAULT_TEMPLATES
template <class BidirectionalIterator, class T, class Reference = T&,
          class Distance = ptrdiff_t>
#else
template <class BidirectionalIterator, class T, class Reference,
          class Distance>
#endif
class reverse_bidirectional_iterator {
  typedef reverse_bidirectional_iterator<BidirectionalIterator, T,
                                         Reference, Distance> self;
protected:
  BidirectionalIterator current;
public:
  typedef bidirectional_iterator_tag     iterator_category;
  typedef T                              value_type;
  typedef Distance                       difference_type;
  typedef T*                             pointer;
  typedef Reference                      reference;

  reverse_bidirectional_iterator() {}
  explicit reverse_bidirectional_iterator(BidirectionalIterator x)
    : current(x) {}
  BidirectionalIterator base() const { return current; }
  Reference operator*() const {
    BidirectionalIterator tmp = current;
    return *--tmp;
  }
#ifndef __SGI_STL_NO_ARROW_OPERATOR
  pointer operator->() const { return &(operator*()); }
#endif /* __SGI_STL_NO_ARROW_OPERATOR */
```

```
  // ++ 變成 --
  self& operator++() {
    --current;
    return *this;
  }
  self operator++(int) {
    self tmp = *this;
    --current;
    return tmp;
  }

  // -- 變成 ++
  self& operator--() {
    ++current;
    return *this;
  }
  self operator--(int) {
    self tmp = *this;
    ++current;
    return tmp;
  }
};

#ifndef __STL_CLASS_PARTIAL_SPECIALIZATION

template <class BidirectionalIterator, class T, class Reference,
          class Distance>
inline bidirectional_iterator_tag
iterator_category(const reverse_bidirectional_iterator<BidirectionalIterator,
                                T,
                                Reference, Distance>&) {
  return bidirectional_iterator_tag();
}

template <class BidirectionalIterator, class T, class Reference,
          class Distance>
inline T*
value_type(const reverse_bidirectional_iterator<BidirectionalIterator, T,
                                      Reference, Distance>&) {
  return (T*) 0;
}

template <class BidirectionalIterator, class T, class Reference,
          class Distance>
inline Distance*
distance_type(const reverse_bidirectional_iterator<BidirectionalIterator, T,
                                        Reference, Distance>&) {
  return (Distance*) 0;
```

```
}

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

template <class BidirectionalIterator, class T, class Reference,
          class Distance>
inline bool operator==(
    const reverse_bidirectional_iterator<BidirectionalIterator, T, Reference,
                                         Distance>& x,
    const reverse_bidirectional_iterator<BidirectionalIterator, T, Reference,
                                         Distance>& y) {
  return x.base() == y.base();
}

#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION

// 以下是C++ 標準所定義的 reverse_iterator，定義於C++ 標準草稿中。它依賴
//  iterator_traits template，並因此依賴partial specialization。
//  先前的那個reverse_bidirectional_iterator 不再是標準草稿的一部份，
//  但仍然保留以備回溯相容。

// 這是一個迭代器配接器（iterator adapter），用來將某個迭代器逆反前進方向，
// 使前進為後退，後退為前進。
template <class Iterator>
class reverse_iterator
{
protected:
  Iterator current;
public:
  typedef typename iterator_traits<Iterator>::iterator_category
          iterator_category;
  typedef typename iterator_traits<Iterator>::value_type
          value_type;
  typedef typename iterator_traits<Iterator>::difference_type
          difference_type;
  typedef typename iterator_traits<Iterator>::pointer
          pointer;
  typedef typename iterator_traits<Iterator>::reference
          reference;

  typedef Iterator iterator_type;
  typedef reverse_iterator<Iterator> self;

public:
  reverse_iterator() {}
  // 下面這個 ctor 將 reverse_iterator 與某個迭代器x 繫結起來。
  explicit reverse_iterator(iterator_type x) : current(x) {}

  reverse_iterator(const self& x) : current(x.current) {}
```

```
#ifdef __STL_MEMBER_TEMPLATES
  template <class Iter>
  reverse_iterator(const reverse_iterator<Iter>& x) : current(x.current) {}
#endif /* __STL_MEMBER_TEMPLATES */

  iterator_type base() const { return current; }
  reference operator*() const {
    Iterator tmp = current;
    return *--tmp;     // 關鍵設計
  }
#ifndef __SGI_STL_NO_ARROW_OPERATOR
  pointer operator->() const { return &(operator*()); }
#endif /* __SGI_STL_NO_ARROW_OPERATOR */

  // ++ 變成 --
  self& operator++() {
    --current;
    return *this;
  }
  self operator++(int) {
    self tmp = *this;
    --current;
    return tmp;
  }
  // -- 變成 ++
  self& operator--() {
    ++current;
    return *this;
  }
  self operator--(int) {
    self tmp = *this;
    ++current;
    return tmp;
  }
  // 前進與後退方向完全逆轉
  self operator+(difference_type n) const {
    return self(current - n);
  }
  self& operator+=(difference_type n) {
    current -= n;
    return *this;
  }
  self operator-(difference_type n) const {
    return self(current + n);
  }
  self& operator-=(difference_type n) {
    current += n;
    return *this;
  }
```

```
  // 注意，下面第一個* 和唯一的 + 都會喚起本類別的 opearator* 和 opreator+，
  // 第二個 * 則不會。（判斷法則：完全看待處理的型別是什麼而定）
  reference operator[](difference_type n) const { return *(*this + n); }
};

template <class Iterator>
inline bool operator==(const reverse_iterator<Iterator>& x,
                       const reverse_iterator<Iterator>& y) {
  return x.base() == y.base();
}

template <class Iterator>
inline bool operator<(const reverse_iterator<Iterator>& x,
                      const reverse_iterator<Iterator>& y) {
  return y.base() < x.base();
}

template <class Iterator>
inline typename reverse_iterator<Iterator>::difference_type
operator-(const reverse_iterator<Iterator>& x,
          const reverse_iterator<Iterator>& y) {
  return y.base() - x.base();
}

template <class Iterator>
inline reverse_iterator<Iterator>
operator+(reverse_iterator<Iterator>::difference_type n,
          const reverse_iterator<Iterator>& x) {
  return reverse_iterator<Iterator>(x.base() - n);
}

#else /* __STL_CLASS_PARTIAL_SPECIALIZATION */

// 下面是舊版的 reverse_iterator，出現於原始的 HP STL 之中。
//  它並不使用 partial specialization.

#ifndef __STL_LIMITED_DEFAULT_TEMPLATES
template <class RandomAccessIterator, class T, class Reference = T&,
          class Distance = ptrdiff_t>
#else
template <class RandomAccessIterator, class T, class Reference,
          class Distance>
#endif
class reverse_iterator {
  typedef reverse_iterator<RandomAccessIterator, T, Reference, Distance>
        self;
protected:
  RandomAccessIterator current;
public:
```

```
  typedef random_access_iterator_tag iterator_category;
  typedef T                       value_type;
  typedef Distance                 difference_type;
  typedef T*                      pointer;
  typedef Reference                reference;

  reverse_iterator() {}
  explicit reverse_iterator(RandomAccessIterator x) : current(x) {}
  RandomAccessIterator base() const { return current; }
  Reference operator*() const { return *(current - 1); }
#ifndef __SGI_STL_NO_ARROW_OPERATOR
  pointer operator->() const { return &(operator*()); }
#endif /* __SGI_STL_NO_ARROW_OPERATOR */
  self& operator++() {
    --current;
    return *this;
  }
  self operator++(int) {
    self tmp = *this;
    --current;
    return tmp;
  }
  self& operator--() {
    ++current;
    return *this;
  }
  self operator--(int) {
    self tmp = *this;
    ++current;
    return tmp;
  }
  self operator+(Distance n) const {
    return self(current - n);
  }
  self& operator+=(Distance n) {
    current -= n;
    return *this;
  }
  self operator-(Distance n) const {
    return self(current + n);
  }
  self& operator-=(Distance n) {
    current += n;
    return *this;
  }
  Reference operator[](Distance n) const { return *(*this + n); }
};

template <class RandomAccessIterator, class T, class Reference, class Distance>
```

```
inline random_access_iterator_tag
iterator_category(const reverse_iterator<RandomAccessIterator, T,
                                  Reference, Distance>&) {
  return random_access_iterator_tag();
}

template <class RandomAccessIterator, class T, class Reference, class Distance>
inline T* value_type(const reverse_iterator<RandomAccessIterator, T,
                                    Reference, Distance>&) {
  return (T*) 0;
}

template <class RandomAccessIterator, class T, class Reference, class Distance>
inline Distance* distance_type(const reverse_iterator<RandomAccessIterator, T,
                                           Reference, Distance>&) {
  return (Distance*) 0;
}

template <class RandomAccessIterator, class T, class Reference, class Distance>
inline bool operator==(const reverse_iterator<RandomAccessIterator, T,
                                     Reference, Distance>& x,
                   const reverse_iterator<RandomAccessIterator, T,
                                     Reference, Distance>& y) {
  return x.base() == y.base();
}

template <class RandomAccessIterator, class T, class Reference, class Distance>
inline bool operator<(const reverse_iterator<RandomAccessIterator, T,
                                    Reference, Distance>& x,
                  const reverse_iterator<RandomAccessIterator, T,
                                    Reference, Distance>& y) {
  return y.base() < x.base();
}

template <class RandomAccessIterator, class T, class Reference, class Distance>
inline Distance operator-(const reverse_iterator<RandomAccessIterator, T,
                                       Reference, Distance>& x,
                      const reverse_iterator<RandomAccessIterator, T,
                                       Reference, Distance>& y) {
  return y.base() - x.base();
}

template <class RandomAccessIter, class T, class Ref, class Dist>
inline reverse_iterator<RandomAccessIter, T, Ref, Dist>
operator+(Dist n, const reverse_iterator<RandomAccessIter, T, Ref, Dist>& x)
{
  return reverse_iterator<RandomAccessIter, T, Ref, Dist>(x.base() - n);
}
```

```
#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

// 這是一個 input iterator，能夠為「來自某一 basic_istream」的物件執行
// 格式化輸入動作。注意，此版本為舊有之 HP 規格，未符合標準介面：
// istream_iterator<T, charT, traits, Distance>
// 然而一般使用 input iterators 時都只使用第一個template 參數，此時以下仍適用。
// 註：SGI STL 3.3 已實作出符合標準介面的 istream_iterator。作法與本版大同小異。
// 本版可讀性較高。
template <class T, class Distance = ptrdiff_t>
class istream_iterator {
  friend bool
  operator== __STL_NULL_TMPL_ARGS (const istream_iterator<T, Distance>& x,
                                   const istream_iterator<T, Distance>& y);
  // 以上語法很奇特，請參考C++ Primer p834: bound friend function template
  // 在 <stl_config.h> 中，__STL_NULL_TMPL_ARGS 被定義為 <>
protected:
  istream* stream;
  T value;
  bool end_marker;
  void read() {
    end_marker = (*stream) ? true : false;
    if (end_marker) *stream >> value;
    // 以上，輸入之後，stream 的狀態可能改變，所以下面再判斷一次以決定 end_marker
    // 當讀到 eof 或讀到型別不符的資料，stream 即處於 false 狀態。
    end_marker = (*stream) ? true : false;
  }
public:
  typedef input_iterator_tag    iterator_category;
  typedef T                     value_type;
  typedef Distance              difference_type;
  typedef const T*              pointer;
  typedef const T&              reference;
  // 以上，因身為input iterator，所以採用 const 比較保險

  // 下面這些ctors 使 istream_iterator 和某個 istream object 繫結起來。
  istream_iterator() : stream(&cin), end_marker(false) {}
  istream_iterator(istream& s) : stream(&s) { read(); }
  // 以上兩行的用法：
  //  istream_iterator<int> eos;          造成 end_marker 為 false。
  //  istream_iterator<int>initer(cin);   引發 read()。程式至此會等待輸入。
  // 因此，下面這兩行客端程式：
  //  istream_iterator<int> initer(cin);      (A)
  //  cout << "please input..." << endl;      (B)
  // 會停留在 (A) 等待一個輸入，然後才執行 (B) 出現提示訊息。這是不合理的現象。
  // 規避之道：永遠在最必要的時候，才定義一個 istream_iterator。

  reference operator*() const { return value; }
#ifndef __SGI_STL_NO_ARROW_OPERATOR
  pointer operator->() const { return &(operator*()); }
```

```
#endif /* __SGI_STL_NO_ARROW_OPERATOR */

  // 迭代器前進一個位置，就代表要讀取一筆資料
  istream_iterator<T, Distance>& operator++() {
    read();
    return *this;
  }
  istream_iterator<T, Distance> operator++(int)  {
    istream_iterator<T, Distance> tmp = *this;
    read();
    return tmp;
  }
};

#ifndef __STL_CLASS_PARTIAL_SPECIALIZATION

template <class T, class Distance>
inline input_iterator_tag
iterator_category(const istream_iterator<T, Distance>&) {
  return input_iterator_tag();
}

template <class T, class Distance>
inline T* value_type(const istream_iterator<T, Distance>&) { return (T*) 0; }

template <class T, class Distance>
inline Distance* distance_type(const istream_iterator<T, Distance>&) {
  return (Distance*) 0;
}

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

template <class T, class Distance>
inline bool operator==(const istream_iterator<T, Distance>& x,
                       const istream_iterator<T, Distance>& y) {
  return x.stream == y.stream && x.end_marker == y.end_marker ||
         x.end_marker == false && y.end_marker == false;
}

// 這是一個 output iterator，能夠將物件格式化輸出到某個 basic_ostream 上。
// 注意，此版本為舊有之 HP 規格，未符合標準介面：
// ostream_iterator<T, charT, traits>
// 然而一般使用 output iterators 時都只使用第一個template 參數，此時以下仍適用。
// 註：SGI STL 3.3 已實作出符合標準介面的 ostream_iterator。作法與本版大同小異。
// 本版可讀性較高。
template <class T>
class ostream_iterator {
protected:
  ostream* stream;
```

```
  const char* string;       // 每次輸出後的間隔符號。
  // 以上注意，可以將變數命名為 string 嗎？可以，但稍後如需使用
  // C++ library string，得寫 std::string.
public:
  typedef output_iterator_tag   iterator_category;
  typedef void                  value_type;
  typedef void                  difference_type;
  typedef void                  pointer;
  typedef void                  reference;

  // 下面這些ctors 使 istream_iterator 和某個 istream object 繫結起來。
  ostream_iterator(ostream& s) : stream(&s), string(0) {}
  ostream_iterator(ostream& s, const char* c) : stream(&s), string(c)  {}
  // 以上 ctors 的用法：
  //  ostream_iterator<int> outiter(cout, ' ');  輸出至 cout，每次間隔一個空格

  // 對迭代器做賦值（assign）動作，就代表要輸出一筆資料
  ostream_iterator<T>& operator=(const T& value) {
    *stream << value;               // 先輸出數值
    if (string) *stream << string;   // 如果狀態無誤，再輸出間隔符號
    return *this;
  }
  ostream_iterator<T>& operator*() { return *this; }
  ostream_iterator<T>& operator++() { return *this; }
  ostream_iterator<T>& operator++(int) { return *this; }
};

#ifndef __STL_CLASS_PARTIAL_SPECIALIZATION

template <class T>
inline output_iterator_tag
iterator_category(const ostream_iterator<T>&) {
  return output_iterator_tag();
}

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

__STL_END_NAMESPACE

#endif /* __SGI_STL_INTERNAL_ITERATOR_H */

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