SGI STL 3.3 **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) 1996-1998
 * 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


template <class _Container>
class back_insert_iterator {
protected:
  _Container* container;
public:
  typedef _Container        container_type;
  typedef output_iterator_tag iterator_category;
  typedef void             value_type;
  typedef void             difference_type;
  typedef void             pointer;
  typedef void             reference;
```

```
  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;
  }
  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 */

template <class _Container>
inline back_insert_iterator<_Container> back_inserter(_Container& __x) {
  return back_insert_iterator<_Container>(__x);
}

template <class _Container>
class front_insert_iterator {
protected:
  _Container* container;
public:
  typedef _Container        container_type;
  typedef output_iterator_tag iterator_category;
  typedef void              value_type;
  typedef void              difference_type;
  typedef void              pointer;
  typedef void              reference;

  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;
  }
  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 */

template <class _Container>
inline front_insert_iterator<_Container> front_inserter(_Container& __x) {
  return front_insert_iterator<_Container>(__x);
}

template <class _Container>
class insert_iterator {
protected:
  _Container* container;
  typename _Container::iterator iter;
public:
  typedef _Container        container_type;
  typedef output_iterator_tag iterator_category;
  typedef void             value_type;
  typedef void             difference_type;
  typedef void             pointer;
  typedef void             reference;

  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;
    return *this;
  }
  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 */

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));
}

#ifndef __STL_LIMITED_DEFAULT_TEMPLATES
template <class _BidirectionalIterator, class _Tp, class _Reference = _Tp&,
          class _Distance = ptrdiff_t>
#else
template <class _BidirectionalIterator, class _Tp, class _Reference,
          class _Distance>
#endif
class reverse_bidirectional_iterator {
  typedef reverse_bidirectional_iterator<_BidirectionalIterator, _Tp,
                                         _Reference, _Distance>  _Self;
protected:
  _BidirectionalIterator current;
public:
  typedef bidirectional_iterator_tag iterator_category;
  typedef _Tp                   value_type;
  typedef _Distance              difference_type;
  typedef _Tp*                  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 */
  _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;
  }
};

#ifndef __STL_CLASS_PARTIAL_SPECIALIZATION

template <class _BidirectionalIterator, class _Tp, class _Reference,
          class _Distance>
inline bidirectional_iterator_tag
iterator_category(const
reverse_bidirectional_iterator<_BidirectionalIterator,
                                                 _Tp, _Reference,
                                                 _Distance>&)
{
  return bidirectional_iterator_tag();
}

template <class _BidirectionalIterator, class _Tp, class _Reference,
          class _Distance>
inline _Tp*
value_type(const reverse_bidirectional_iterator<_BidirectionalIterator, _Tp,
                                          _Reference, _Distance>&)
{
  return (_Tp*) 0;
}

template <class _BidirectionalIterator, class _Tp, class _Reference,
          class _Distance>
inline _Distance*
distance_type(const reverse_bidirectional_iterator<_BidirectionalIterator,
                                                 _Tp,
                                                 _Reference, _Distance>&)
{
  return (_Distance*) 0;
}

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

template <class _BiIter, class _Tp, class _Ref, class _Distance>
```

```
inline bool operator==(
    const reverse_bidirectional_iterator<_BiIter, _Tp, _Ref, _Distance>& __x,
    const reverse_bidirectional_iterator<_BiIter, _Tp, _Ref, _Distance>& __y)
{
  return __x.base() == __y.base();
}

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

template <class _BiIter, class _Tp, class _Ref, class _Distance>
inline bool operator!=(
    const reverse_bidirectional_iterator<_BiIter, _Tp, _Ref, _Distance>& __x,
    const reverse_bidirectional_iterator<_BiIter, _Tp, _Ref, _Distance>& __y)
{
  return !(__x == __y);
}

#endif /* __STL_FUNCTION_TMPL_PARTIAL_ORDER */


#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION

// This is the new version of reverse_iterator, as defined in the
//  draft C++ standard.  It relies on the iterator_traits template,
//  which in turn relies on partial specialization.  The class
//  reverse_bidirectional_iterator is no longer part of the draft
//  standard, but it is retained for backward compatibility.

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() {}
  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.base()) {}
#endif /* __STL_MEMBER_TEMPLATES */

  iterator_type base() const { return current; }
  reference operator*() const {
    _Iterator __tmp = current;
    return *--__tmp;
  }
#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+(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;
  }
  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();
}

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

template <class _Iterator>
inline bool operator!=(const reverse_iterator<_Iterator>& __x,
                       const reverse_iterator<_Iterator>& __y) {
  return !(__x == __y);
}

template <class _Iterator>
inline bool operator>(const reverse_iterator<_Iterator>& __x,
                      const reverse_iterator<_Iterator>& __y) {
  return __y < __x;
}

template <class _Iterator>
inline bool operator<=(const reverse_iterator<_Iterator>& __x,
                       const reverse_iterator<_Iterator>& __y) {
  return !(__y < __x);
}

template <class _Iterator>
inline bool operator>=(const reverse_iterator<_Iterator>& __x,
                      const reverse_iterator<_Iterator>& __y) {
  return !(__x < __y);
}

#endif /* __STL_FUNCTION_TMPL_PARTIAL_ORDER */

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+(typename reverse_iterator<_Iterator>::difference_type __n,
        const reverse_iterator<_Iterator>& __x) {
  return reverse_iterator<_Iterator>(__x.base() - __n);
}

#else /* __STL_CLASS_PARTIAL_SPECIALIZATION */

// This is the old version of reverse_iterator, as found in the original
//  HP STL.  It does not use partial specialization.

#ifndef __STL_LIMITED_DEFAULT_TEMPLATES
template <class _RandomAccessIterator, class _Tp, class _Reference = _Tp&,
        class _Distance = ptrdiff_t>
#else
template <class _RandomAccessIterator, class _Tp, class _Reference,
        class _Distance>
#endif
class reverse_iterator {
  typedef reverse_iterator<_RandomAccessIterator, _Tp, _Reference, _Distance>
       _Self;
protected:
  _RandomAccessIterator current;
public:
  typedef random_access_iterator_tag iterator_category;
  typedef _Tp                    value_type;
  typedef _Distance               difference_type;
  typedef _Tp*                   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 _Tp,
          class _Reference, class _Distance>
inline random_access_iterator_tag
iterator_category(const reverse_iterator<_RandomAccessIterator, _Tp,
                                         _Reference, _Distance>&)
{
  return random_access_iterator_tag();
}

template <class _RandomAccessIterator, class _Tp,
          class _Reference, class _Distance>
inline _Tp* value_type(const reverse_iterator<_RandomAccessIterator, _Tp,
                                              _Reference, _Distance>&)
{
  return (_Tp*) 0;
}

template <class _RandomAccessIterator, class _Tp,
          class _Reference, class _Distance>
inline _Distance*
distance_type(const reverse_iterator<_RandomAccessIterator,
```

```
                                _Tp, _Reference, _Distance>&)
{
  return (_Distance*) 0;
}


template <class _RandomAccessIterator, class _Tp,
          class _Reference, class _Distance>
inline bool
operator==(const reverse_iterator<_RandomAccessIterator, _Tp,
                                  _Reference, _Distance>& __x,
           const reverse_iterator<_RandomAccessIterator, _Tp,
                                  _Reference, _Distance>& __y)
{
  return __x.base() == __y.base();
}

template <class _RandomAccessIterator, class _Tp,
          class _Reference, class _Distance>
inline bool
operator<(const reverse_iterator<_RandomAccessIterator, _Tp,
                                 _Reference, _Distance>& __x,
          const reverse_iterator<_RandomAccessIterator, _Tp,
                                 _Reference, _Distance>& __y)
{
  return __y.base() < __x.base();
}

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

template <class _RandomAccessIterator, class _Tp,
          class _Reference, class _Distance>
inline bool
operator!=(const reverse_iterator<_RandomAccessIterator, _Tp,
                                  _Reference, _Distance>& __x,
           const reverse_iterator<_RandomAccessIterator, _Tp,
                                  _Reference, _Distance>& __y) {
  return !(__x == __y);
}

template <class _RandomAccessIterator, class _Tp,
          class _Reference, class _Distance>
inline bool
operator>(const reverse_iterator<_RandomAccessIterator, _Tp,
                                 _Reference, _Distance>& __x,
          const reverse_iterator<_RandomAccessIterator, _Tp,
                                 _Reference, _Distance>& __y) {
  return __y < __x;
}
```

```
template <class _RandomAccessIterator, class _Tp,
          class _Reference, class _Distance>
inline bool
operator<=(const reverse_iterator<_RandomAccessIterator, _Tp,
                                  _Reference, _Distance>& __x,
           const reverse_iterator<_RandomAccessIterator, _Tp,
                                  _Reference, _Distance>& __y) {
  return !(__y < __x);
}

template <class _RandomAccessIterator, class _Tp,
          class _Reference, class _Distance>
inline bool
operator>=(const reverse_iterator<_RandomAccessIterator, _Tp,
                                  _Reference, _Distance>& __x,
           const reverse_iterator<_RandomAccessIterator, _Tp,
                                  _Reference, _Distance>& __y) {
  return !(__x < __y);
}

#endif /* __STL_FUNCTION_TMPL_PARTIAL_ORDER */

template <class _RandomAccessIterator, class _Tp,
          class _Reference, class _Distance>
inline _Distance
operator-(const reverse_iterator<_RandomAccessIterator, _Tp,
                                 _Reference, _Distance>& __x,
          const reverse_iterator<_RandomAccessIterator, _Tp,
                                 _Reference, _Distance>& __y)
{
  return __y.base() - __x.base();
}

template <class _RandAccIter, class _Tp, class _Ref, class _Dist>
inline reverse_iterator<_RandAccIter, _Tp, _Ref, _Dist>
operator+(_Dist __n,
          const reverse_iterator<_RandAccIter, _Tp, _Ref, _Dist>& __x)
{
  return reverse_iterator<_RandAccIter, _Tp, _Ref, _Dist>(__x.base() - __n);
}

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

// istream_iterator and ostream_iterator look very different if we're
// using new, templatized iostreams than if we're using the old cfront
// version.

#ifdef __STL_USE_NEW_IOSTREAMS
```

```
template <class _Tp,
          class _CharT = char, class _Traits = char_traits<_CharT>,
          class _Dist = ptrdiff_t>
class istream_iterator {
public:
  typedef _CharT                     char_type;
  typedef _Traits                    traits_type;
  typedef basic_istream<_CharT, _Traits> istream_type;

  typedef input_iterator_tag           iterator_category;
  typedef _Tp                        value_type;
  typedef _Dist                      difference_type;
  typedef const _Tp*                  pointer;
  typedef const _Tp&                  reference;

  istream_iterator() : _M_stream(0), _M_ok(false) {}
  istream_iterator(istream_type& __s) : _M_stream(&__s) { _M_read(); }

  reference operator*() const { return _M_value; }
  pointer operator->() const { return &(operator*()); }

  istream_iterator& operator++() {
    _M_read();
    return *this;
  }
  istream_iterator operator++(int)  {
    istream_iterator __tmp = *this;
    _M_read();
    return __tmp;
  }

  bool _M_equal(const istream_iterator& __x) const
    { return (_M_ok == __x._M_ok) && (!_M_ok || _M_stream == __x._M_stream); }

private:
  istream_type* _M_stream;
  _Tp _M_value;
  bool _M_ok;

  void _M_read() {
    _M_ok = (_M_stream && *_M_stream) ? true : false;
    if (_M_ok) {
      *_M_stream >> _M_value;
      _M_ok = *_M_stream ? true : false;
    }
  }
};
```

```
template <class _Tp, class _CharT, class _Traits, class _Dist>
inline bool
operator==(const istream_iterator<_Tp, _CharT, _Traits, _Dist>& __x,
         const istream_iterator<_Tp, _CharT, _Traits, _Dist>& __y) {
  return __x._M_equal(__y);
}

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

template <class _Tp, class _CharT, class _Traits, class _Dist>
inline bool
operator!=(const istream_iterator<_Tp, _CharT, _Traits, _Dist>& __x,
         const istream_iterator<_Tp, _CharT, _Traits, _Dist>& __y) {
  return !__x._M_equal(__y);
}

#endif /* __STL_FUNCTION_TMPL_PARTIAL_ORDER */

template <class _Tp,
        class _CharT = char, class _Traits = char_traits<_CharT> >
class ostream_iterator {
public:
  typedef _CharT                    char_type;
  typedef _Traits                   traits_type;
  typedef basic_ostream<_CharT, _Traits> ostream_type;

  typedef output_iterator_tag         iterator_category;
  typedef void                      value_type;
  typedef void                      difference_type;
  typedef void                      pointer;
  typedef void                      reference;

  ostream_iterator(ostream_type& __s) : _M_stream(&__s), _M_string(0) {}
  ostream_iterator(ostream_type& __s, const _CharT* __c)
    : _M_stream(&__s), _M_string(__c)  {}
  ostream_iterator<_Tp>& operator=(const _Tp& __value) {
    *_M_stream << __value;
    if (_M_string) *_M_stream << _M_string;
    return *this;
  }
  ostream_iterator<_Tp>& operator*() { return *this; }
  ostream_iterator<_Tp>& operator++() { return *this; }
  ostream_iterator<_Tp>& operator++(int) { return *this; }
private:
  ostream_type* _M_stream;
  const _CharT* _M_string;
};

// The default template argument is declared in iosfwd
```

```
// We do not read any characters until operator* is called.  The first
// time operator* is called, it calls getc.  Subsequent calls to getc
// return a cached character, and calls to operator++ use snextc.  Before
// operator* or operator++ has been called, _M_is_initialized is false.
template<class _CharT, class _Traits>
class istreambuf_iterator
  : public iterator<input_iterator_tag, _CharT,
                    typename _Traits::off_type, _CharT*, _CharT&>
{
public:
  typedef _CharT                       char_type;
  typedef _Traits                      traits_type;
  typedef typename _Traits::int_type      int_type;
  typedef basic_streambuf<_CharT, _Traits> streambuf_type;
  typedef basic_istream<_CharT, _Traits>   istream_type;

public:
  istreambuf_iterator(streambuf_type* __p = 0) { this->_M_init(__p); }
  istreambuf_iterator(istream_type& __is) { this->_M_init(__is.rdbuf()); }

  char_type operator*() const
    { return _M_is_initialized ? _M_c : _M_dereference_aux(); }

  istreambuf_iterator& operator++() { this->_M_nextc(); return *this; }
  istreambuf_iterator  operator++(int) {
    if (!_M_is_initialized)
      _M_postincr_aux();
    istreambuf_iterator __tmp = *this;
    this->_M_nextc();
    return __tmp;
  }

  bool equal(const istreambuf_iterator& __i) const {
    return this->_M_is_initialized && __i._M_is_initialized
      ? this->_M_eof == __i._M_eof
      : this->_M_equal_aux(__i);
  }

private:
  void _M_init(streambuf_type* __p) {
    _M_buf = __p;
    _M_eof = !__p;
    _M_is_initialized = _M_eof;
  }

  char_type _M_dereference_aux() const;
  bool _M_equal_aux(const istreambuf_iterator&) const;
  void _M_postincr_aux();
```

```
  void _M_nextc() {
    int_type __c = _M_buf->snextc();
    _M_c = traits_type::to_char_type(__c);
    _M_eof = traits_type::eq_int_type(__c, traits_type::eof());
    _M_is_initialized = true;
  }

  void _M_getc() const {
    int_type __c = _M_buf->sgetc();
    _M_c = traits_type::to_char_type(__c);
    _M_eof = traits_type::eq_int_type(__c, traits_type::eof());
    _M_is_initialized = true;
  }

private:
  streambuf_type* _M_buf;
  mutable _CharT _M_c;
  mutable bool _M_eof : 1;
  mutable bool _M_is_initialized : 1;
};

template<class _CharT, class _Traits>
_CharT istreambuf_iterator<_CharT, _Traits>::_M_dereference_aux() const
{
  this->_M_getc();
  return _M_c;
}

template<class _CharT, class _Traits>
bool istreambuf_iterator<_CharT, _Traits>
  ::_M_equal_aux(const istreambuf_iterator& __i) const
{
  if (!this->_M_is_initialized)
    this->_M_getc();
  if (!__i._M_is_initialized)
    __i._M_getc();

  return this->_M_eof == __i._M_eof;
}

template<class _CharT, class _Traits>
void istreambuf_iterator<_CharT, _Traits>::_M_postincr_aux()
{
  this->_M_getc();
}

template<class _CharT, class _Traits>
inline bool operator==(const istreambuf_iterator<_CharT, _Traits>& __x,
```

```
                    const istreambuf_iterator<_CharT, _Traits>& __y) {
  return __x.equal(__y);
}

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

template<class _CharT, class _Traits>
inline bool operator!=(const istreambuf_iterator<_CharT, _Traits>& __x,
                    const istreambuf_iterator<_CharT, _Traits>& __y) {
  return !__x.equal(__y);
}

#endif /* __STL_FUNCTION_TMPL_PARTIAL_ORDER */

// The default template argument is declared in iosfwd
template<class _CharT, class _Traits>
class ostreambuf_iterator
  : public iterator<output_iterator_tag, void, void, void, void>
{
public:
  typedef _CharT                      char_type;
  typedef _Traits                     traits_type;
  typedef typename _Traits::int_type      int_type;
  typedef basic_streambuf<_CharT, _Traits> streambuf_type;
  typedef basic_ostream<_CharT, _Traits>   ostream_type;

public:
  ostreambuf_iterator(streambuf_type* __buf) : _M_buf(__buf), _M_ok(__buf) {}
  ostreambuf_iterator(ostream_type& __o)
    : _M_buf(__o.rdbuf()), _M_ok(__o.rdbuf() != 0) {}

  ostreambuf_iterator& operator=(char_type __c) {
    _M_ok = _M_ok && !traits_type::eq_int_type(_M_buf->sputc(__c),
                                       traits_type::eof());
    return *this;
  }

  ostreambuf_iterator& operator*()    { return *this; }
  ostreambuf_iterator& operator++()   { return *this; }
  ostreambuf_iterator& operator++(int) { return *this; }

  bool failed() const { return !_M_ok; }

private:
  streambuf_type* _M_buf;
  bool _M_ok;
};

#else /* __STL_USE_NEW_IOSTREAMS */
```

```
template <class _Tp, class _Dist = ptrdiff_t> class istream_iterator;

template <class _Tp, class _Dist>
inline bool operator==(const istream_iterator<_Tp, _Dist>&,
                       const istream_iterator<_Tp, _Dist>&);

template <class _Tp, class _Dist>
class istream_iterator {
#ifdef __STL_TEMPLATE_FRIENDS
  template <class _T1, class _D1>
  friend bool operator==(const istream_iterator<_T1, _D1>&,
                         const istream_iterator<_T1, _D1>&);
#else /* __STL_TEMPLATE_FRIENDS */
  friend bool __STD_QUALIFIER
  operator== __STL_NULL_TMPL_ARGS (const istream_iterator&,
                                   const istream_iterator&);
#endif /* __STL_TEMPLATE_FRIENDS */

protected:
  istream* _M_stream;
  _Tp _M_value;
  bool _M_end_marker;
  void _M_read() {
    _M_end_marker = (*_M_stream) ? true : false;
    if (_M_end_marker) *_M_stream >> _M_value;
    _M_end_marker = (*_M_stream) ? true : false;
  }
public:
  typedef input_iterator_tag  iterator_category;
  typedef _Tp                 value_type;
  typedef _Dist               difference_type;
  typedef const _Tp*          pointer;
  typedef const _Tp&          reference;

  istream_iterator() : _M_stream(&cin), _M_end_marker(false) {}
  istream_iterator(istream& __s) : _M_stream(&__s) { _M_read(); }
  reference operator*() const { return _M_value; }
#ifndef __SGI_STL_NO_ARROW_OPERATOR
  pointer operator->() const { return &(operator*()); }
#endif /* __SGI_STL_NO_ARROW_OPERATOR */
  istream_iterator<_Tp, _Dist>& operator++() {
    _M_read();
    return *this;
  }
  istream_iterator<_Tp, _Dist> operator++(int)  {
    istream_iterator<_Tp, _Dist> __tmp = *this;
    _M_read();
    return __tmp;
```

```
  }
};

#ifndef __STL_CLASS_PARTIAL_SPECIALIZATION

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

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

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

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

template <class _Tp, class _Distance>
inline bool operator==(const istream_iterator<_Tp, _Distance>& __x,
                       const istream_iterator<_Tp, _Distance>& __y) {
  return (__x._M_stream == __y._M_stream &&
          __x._M_end_marker == __y._M_end_marker) ||
         __x._M_end_marker == false && __y._M_end_marker == false;
}

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

template <class _Tp, class _Distance>
inline bool operator!=(const istream_iterator<_Tp, _Distance>& __x,
                       const istream_iterator<_Tp, _Distance>& __y) {
  return !(__x == __y);
}

#endif /* __STL_FUNCTION_TMPL_PARTIAL_ORDER */

template <class _Tp>
class ostream_iterator {
protected:
  ostream* _M_stream;
  const char* _M_string;
public:
  typedef output_iterator_tag iterator_category;
  typedef void             value_type;
  typedef void             difference_type;
```

```
  typedef void            pointer;
  typedef void            reference;

  ostream_iterator(ostream& __s) : _M_stream(&__s), _M_string(0) {}
  ostream_iterator(ostream& __s, const char* __c)
    : _M_stream(&__s), _M_string(__c)  {}
  ostream_iterator<_Tp>& operator=(const _Tp& __value) {
    *_M_stream << __value;
    if (_M_string) *_M_stream << _M_string;
    return *this;
  }
  ostream_iterator<_Tp>& operator*() { return *this; }
  ostream_iterator<_Tp>& operator++() { return *this; }
  ostream_iterator<_Tp>& operator++(int) { return *this; }
};

#ifndef __STL_CLASS_PARTIAL_SPECIALIZATION

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

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

#endif /* __STL_USE_NEW_IOSTREAMS */

__STL_END_NAMESPACE

#endif /* __SGI_STL_INTERNAL_ITERATOR_H */

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