G++ 2.95 for Solaris，\g003\sgi-stl-of-gcc295-for-solaris**bitset 完整列表**

```
/*
 * Copyright (c) 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.
 */

#ifndef __SGI_STL_BITSET
#define __SGI_STL_BITSET

// This implementation of bitset<> has a second template parameter,
// _WordT, which defaults to unsigned long.  *YOU SHOULD NOT USE
// THIS FEATURE*.  It is experimental, and it may be removed in
// future releases.

// A bitset of size N, using words of type _WordT, will have
// N % (sizeof(_WordT) * CHAR_BIT) unused bits.  (They are the high-
// order bits in the highest word.)  It is a class invariant
// of class bitset<> that those unused bits are always zero.

// Most of the actual code isn't contained in bitset<> itself, but in the
// base class _Base_bitset.  The base class works with whole words, not with
// individual bits.  This allows us to specialize _Base_bitset for the
// important special case where the bitset is only a single word.

// The C++ standard does not define the precise semantics of operator[].
// In this implementation the const version of operator[] is equivalent
// to test(), except that it does no range checking.  The non-const version
// returns a reference to a bit, again without doing any range checking.


#include <stddef.h>    // for size_t
#include <string>
#include <stdexcept>   // for invalid_argument, out_of_range, overflow_error
#include <iostream.h>  // for istream, ostream

#define __BITS_PER_WORDT(__wt) (CHAR_BIT*sizeof(__wt))
#define __BITSET_WORDS(__n,__wt) \
 ((__n) < 1 ? 1 : ((__n) + __BITS_PER_WORDT(__wt) - 1)/__BITS_PER_WORDT(__wt))

__STL_BEGIN_NAMESPACE
```

```
#if defined(__sgi) && !defined(__GNUC__) && (_MIPS_SIM != _MIPS_SIM_ABI32)
#pragma set woff 1209
#endif

// structure to aid in counting bits
template<bool __dummy>
struct _Bit_count {
  static unsigned char _S_bit_count[256];
};

// Mapping from 8 bit unsigned integers to the index of the first one
// bit:
template<bool __dummy>
struct _First_one {
  static unsigned char _S_first_one[256];
};

//
// Base class: general case.
//

template<size_t _Nw, class _WordT>
struct _Base_bitset {
  _WordT _M_w[_Nw];             // 0 is the least significant word.

  _Base_bitset( void ) { _M_do_reset(); }

  _Base_bitset(unsigned long __val);

  static size_t _S_whichword( size_t __pos ) {
    return __pos / __BITS_PER_WORDT(_WordT);
  }
  static size_t _S_whichbyte( size_t __pos ) {
    return (__pos % __BITS_PER_WORDT(_WordT)) / CHAR_BIT;
  }
  static size_t _S_whichbit( size_t __pos ) {
    return __pos % __BITS_PER_WORDT(_WordT);
  }
  static _WordT _S_maskbit( size_t __pos ) {
    return (static_cast<_WordT>(1)) << _S_whichbit(__pos);
  }

  _WordT& _M_getword(size_t __pos)      { return _M_w[_S_whichword(__pos)]; }
  _WordT  _M_getword(size_t __pos) const { return _M_w[_S_whichword(__pos)]; }

  _WordT& _M_hiword()      { return _M_w[_Nw - 1]; }
  _WordT  _M_hiword() const { return _M_w[_Nw - 1]; }

  void _M_do_and(const _Base_bitset<_Nw,_WordT>& __x) {
```

```
  for ( size_t __i = 0; __i < _Nw; __i++ ) {
    _M_w[__i] &= __x._M_w[__i];
  }
}

void _M_do_or(const _Base_bitset<_Nw,_WordT>& __x) {
  for ( size_t __i = 0; __i < _Nw; __i++ ) {
    _M_w[__i] |= __x._M_w[__i];
  }
}

void _M_do_xor(const _Base_bitset<_Nw,_WordT>& __x) {
  for ( size_t __i = 0; __i < _Nw; __i++ ) {
    _M_w[__i] ^= __x._M_w[__i];
  }
}

void _M_do_left_shift(size_t __shift);

void _M_do_right_shift(size_t __shift);

void _M_do_flip() {
  for ( size_t __i = 0; __i < _Nw; __i++ ) {
    _M_w[__i] = ~_M_w[__i];
  }
}

void _M_do_set() {
  for ( size_t __i = 0; __i < _Nw; __i++ ) {
    _M_w[__i] = ~static_cast<_WordT>(0);
  }
}

void _M_do_reset() {
  for ( size_t __i = 0; __i < _Nw; __i++ ) {
    _M_w[__i] = 0;
  }
}

bool _M_is_equal(const _Base_bitset<_Nw,_WordT>& __x) const {
  for (size_t __i = 0; __i < _Nw; ++__i) {
    if (_M_w[__i] != __x._M_w[__i])
      return false;
  }
  return true;
}

bool _M_is_any() const {
  for ( size_t __i = 0; __i < __BITSET_WORDS(_Nw,_WordT); __i++ ) {
```

```
      if ( _M_w[__i] != static_cast<_WordT>(0) )
        return true;
    }
    return false;
  }

  size_t _M_do_count() const {
    size_t __result = 0;
    const unsigned char* __byte_ptr = (const unsigned char*)_M_w;
    const unsigned char* __end_ptr = (const unsigned char*)(_M_w+_Nw);

    while ( __byte_ptr < __end_ptr ) {
      __result += _Bit_count<true>::_S_bit_count[*__byte_ptr];
      __byte_ptr++;
    }
    return __result;
  }

  unsigned long _M_do_to_ulong() const;

  // find first "on" bit
  size_t _M_do_find_first(size_t __not_found) const;

  // find the next "on" bit that follows "prev"
  size_t _M_do_find_next(size_t __prev, size_t __not_found) const;
};

//
// Definitions of non-inline functions from _Base_bitset.
//

template<size_t _Nw, class _WordT>
_Base_bitset<_Nw, _WordT>::_Base_bitset(unsigned long __val)
{
  _M_do_reset();
  const size_t __n = min(sizeof(unsigned long)*CHAR_BIT,
                         __BITS_PER_WORDT(_WordT)*_Nw);
  for(size_t __i = 0; __i < __n; ++__i, __val >>= 1)
    if ( __val & 0x1 )
      _M_getword(__i) |= _S_maskbit(__i);
}

template<size_t _Nw, class _WordT>
void _Base_bitset<_Nw, _WordT>::_M_do_left_shift(size_t __shift)
{
  if (__shift != 0) {
    const size_t __wshift = __shift / __BITS_PER_WORDT(_WordT);
    const size_t __offset = __shift % __BITS_PER_WORDT(_WordT);
    const size_t __sub_offset = __BITS_PER_WORDT(_WordT) - __offset;
```

```
    size_t __n = _Nw - 1;
    for ( ; __n > __wshift; --__n)
      _M_w[__n] = (_M_w[__n - __wshift] << __offset) |
                  (_M_w[__n - __wshift - 1] >> __sub_offset);
    if (__n == __wshift)
      _M_w[__n] = _M_w[0] << __offset;
    for (size_t __n1 = 0; __n1 < __n; ++__n1)
      _M_w[__n1] = static_cast<_WordT>(0);
  }
}

template<size_t _Nw, class _WordT>
void _Base_bitset<_Nw, _WordT>::_M_do_right_shift(size_t __shift)
{
  if (__shift != 0) {
    const size_t __wshift = __shift / __BITS_PER_WORDT(_WordT);
    const size_t __offset = __shift % __BITS_PER_WORDT(_WordT);
    const size_t __sub_offset = __BITS_PER_WORDT(_WordT) - __offset;
    const size_t __limit = _Nw - __wshift - 1;
    size_t __n = 0;
    for ( ; __n < __limit; ++__n)
      _M_w[__n] = (_M_w[__n + __wshift] >> __offset) |
                  (_M_w[__n + __wshift + 1] << __sub_offset);
    _M_w[__limit] = _M_w[_Nw-1] >> __offset;
    for (size_t __n1 = __limit + 1; __n1 < _Nw; ++__n1)
      _M_w[__n1] = static_cast<_WordT>(0);
  }
}

template<size_t _Nw, class _WordT>
unsigned long _Base_bitset<_Nw, _WordT>::_M_do_to_ulong() const
{
  const overflow_error __overflow("bitset");

  if (sizeof(_WordT) >= sizeof(unsigned long)) {
    for (size_t __i = 1; __i < _Nw; ++__i)
      if (_M_w[__i])
        __STL_THROW(__overflow);

    const _WordT __mask = static_cast<_WordT>(static_cast<unsigned long>(-1));
    if (_M_w[0] & ~__mask)
      __STL_THROW(__overflow);

    return static_cast<unsigned long>(_M_w[0] & __mask);
  }
  else {                    // sizeof(_WordT) < sizeof(unsigned long).
    const size_t __nwords =
      (sizeof(unsigned long) + sizeof(_WordT) - 1) / sizeof(_WordT);
```

```
    size_t __min_nwords = __nwords;
    if (_Nw > __nwords) {
      for (size_t __i = __nwords; __i < _Nw; ++__i)
        if (_M_w[__i])
          __STL_THROW(__overflow);
    }
    else
      __min_nwords = _Nw;

    // If unsigned long is 8 bytes and _WordT is 6 bytes, then an unsigned
    // long consists of all of one word plus 2 bytes from another word.
    const size_t __part = sizeof(unsigned long) % sizeof(_WordT);

    if (__part != 0 && __nwords <= _Nw &&
        (_M_w[__min_nwords - 1] >> ((sizeof(_WordT) - __part) * CHAR_BIT)) != 0)
      __STL_THROW(__overflow);

    unsigned long __result = 0;
    for (size_t __i = 0; __i < __min_nwords; ++__i) {
      __result |= static_cast<unsigned long>(
         _M_w[__i]) << (__i * sizeof(_WordT) * CHAR_BIT);
    }
    return __result;
  }
} // End _M_do_to_ulong

template<size_t _Nw, class _WordT>
size_t _Base_bitset<_Nw, _WordT>::_M_do_find_first(size_t __not_found) const
{
  for ( size_t __i = 0; __i < _Nw; __i++ ) {
    _WordT __thisword = _M_w[__i];
    if ( __thisword != static_cast<_WordT>(0) ) {
      // find byte within word
      for ( size_t __j = 0; __j < sizeof(_WordT); __j++ ) {
        unsigned char __this_byte
          = static_cast<unsigned char>(__thisword & (~(unsigned char)0));
        if ( __this_byte )
          return __i*__BITS_PER_WORDT(_WordT) + __j*CHAR_BIT +
            _First_one<true>::_S_first_one[__this_byte];

        __thisword >>= CHAR_BIT;
      }
    }
  }
  // not found, so return an indication of failure.
  return __not_found;
}

template<size_t _Nw, class _WordT>
```

```
size_t
_Base_bitset<_Nw, _WordT>::_M_do_find_next(size_t __prev,
                                           size_t __not_found) const
{
  // make bound inclusive
  ++__prev;

  // check out of bounds
  if ( __prev >= _Nw * __BITS_PER_WORDT(_WordT) )
    return __not_found;

    // search first word
  size_t __i = _S_whichword(__prev);
  _WordT __thisword = _M_w[__i];

    // mask off bits below bound
  __thisword &= (~static_cast<_WordT>(0)) << _S_whichbit(__prev);

  if ( __thisword != static_cast<_WordT>(0) ) {
    // find byte within word
    // get first byte into place
    __thisword >>= _S_whichbyte(__prev) * CHAR_BIT;
    for ( size_t __j = _S_whichbyte(__prev); __j < sizeof(_WordT); __j++ ) {
      unsigned char __this_byte
        = static_cast<unsigned char>(__thisword & (~(unsigned char)0));
      if ( __this_byte )
        return __i*__BITS_PER_WORDT(_WordT) + __j*CHAR_BIT +
          _First_one<true>::_S_first_one[__this_byte];

      __thisword >>= CHAR_BIT;
    }
  }

  // check subsequent words
  __i++;
  for ( ; __i < _Nw; __i++ ) {
    _WordT __thisword = _M_w[__i];
    if ( __thisword != static_cast<_WordT>(0) ) {
      // find byte within word
      for ( size_t __j = 0; __j < sizeof(_WordT); __j++ ) {
        unsigned char __this_byte
          = static_cast<unsigned char>(__thisword & (~(unsigned char)0));
        if ( __this_byte )
          return __i*__BITS_PER_WORDT(_WordT) + __j*CHAR_BIT +
            _First_one<true>::_S_first_one[__this_byte];

        __thisword >>= CHAR_BIT;
      }
    }
```

```
  }

  // not found, so return an indication of failure.
  return __not_found;
} // end _M_do_find_next


// ------------------------------------------------------------

//
// Base class: specialization for a single word.
//

template<class _WordT>
struct _Base_bitset<1, _WordT> {
  _WordT _M_w;

  _Base_bitset( void ) { _M_do_reset(); }

  _Base_bitset(unsigned long __val);

  static size_t _S_whichword( size_t __pos ) {
    return __pos / __BITS_PER_WORDT(_WordT);
  }
  static size_t _S_whichbyte( size_t __pos ) {
    return (__pos % __BITS_PER_WORDT(_WordT)) / CHAR_BIT;
  }
  static size_t _S_whichbit( size_t __pos ) {
    return __pos % __BITS_PER_WORDT(_WordT);
  }
  static _WordT _S_maskbit( size_t __pos ) {
    return (static_cast<_WordT>(1)) << _S_whichbit(__pos);
  }

  _WordT& _M_getword(size_t)      { return _M_w; }
  _WordT  _M_getword(size_t) const { return _M_w; }

  _WordT& _M_hiword()      { return _M_w; }
  _WordT  _M_hiword() const { return _M_w; }

  void _M_do_and(const _Base_bitset<1,_WordT>& __x) { _M_w &= __x._M_w; }
  void _M_do_or(const _Base_bitset<1,_WordT>& __x)  { _M_w |= __x._M_w; }
  void _M_do_xor(const _Base_bitset<1,_WordT>& __x) { _M_w ^= __x._M_w; }
  void _M_do_left_shift(size_t __shift)    { _M_w <<= __shift; }
  void _M_do_right_shift(size_t __shift)   { _M_w >>= __shift; }
  void _M_do_flip()                   { _M_w = ~_M_w; }
  void _M_do_set()                    { _M_w = ~static_cast<_WordT>(0); }
  void _M_do_reset()                  { _M_w = 0; }
```

```
  bool _M_is_equal(const _Base_bitset<1,_WordT>& __x) const {
    return _M_w == __x._M_w;
  }
  bool _M_is_any() const {
    return _M_w != 0;
  }

  size_t _M_do_count() const {
    size_t __result = 0;
    const unsigned char* __byte_ptr = (const unsigned char*)&_M_w;
    const unsigned char* __end_ptr = ((const unsigned char*)&_M_w)+sizeof(_M_w);
    while ( __byte_ptr < __end_ptr ) {
      __result += _Bit_count<true>::_S_bit_count[*__byte_ptr];
      __byte_ptr++;
    }
    return __result;
  }

  unsigned long _M_do_to_ulong() const {
    if (sizeof(_WordT) <= sizeof(unsigned long))
       return static_cast<unsigned long>(_M_w);
    else {
      const _WordT __mask = static_cast<_WordT>(static_cast<unsigned long>(-1));
      if (_M_w & ~__mask)
        __STL_THROW(overflow_error("bitset"));
      return static_cast<unsigned long>(_M_w);
    }
  }

  size_t _M_do_find_first(size_t __not_found) const;

  // find the next "on" bit that follows "prev"
  size_t _M_do_find_next(size_t __prev, size_t __not_found) const;

};

//
// Definitions of non-inline functions from the single-word version of
//  _Base_bitset.
//

template <class _WordT>
_Base_bitset<1, _WordT>::_Base_bitset(unsigned long __val)
{
  _M_do_reset();
  const size_t __n = min(sizeof(unsigned long)*CHAR_BIT,
                         __BITS_PER_WORDT(_WordT)*_Nw);
  for(size_t __i = 0; __i < __n; ++__i, __val >>= 1)
    if ( __val & 0x1 )
```

```
      _M_w |= _S_maskbit(__i);
}

template <class _WordT>
size_t _Base_bitset<1, _WordT>::_M_do_find_first(size_t __not_found) const
{
  _WordT __thisword = _M_w;

  if ( __thisword != static_cast<_WordT>(0) ) {
    // find byte within word
    for ( size_t __j = 0; __j < sizeof(_WordT); __j++ ) {
      unsigned char __this_byte
        = static_cast<unsigned char>(__thisword & (~(unsigned char)0));
      if ( __this_byte )
        return __j*CHAR_BIT + _First_one<true>::_S_first_one[__this_byte];

      __thisword >>= CHAR_BIT;
    }
  }
  // not found, so return a value that indicates failure.
  return __not_found;
}

template <class _WordT>
size_t
_Base_bitset<1, _WordT>::_M_do_find_next(size_t __prev,
                                         size_t __not_found ) const
{
  // make bound inclusive
  ++__prev;

  // check out of bounds
  if ( __prev >= __BITS_PER_WORDT(_WordT) )
    return __not_found;

    // search first (and only) word
  _WordT __thisword = _M_w;

  // mask off bits below bound
  __thisword &= (~static_cast<_WordT>(0)) << _S_whichbit(__prev);

  if ( __thisword != static_cast<_WordT>(0) ) {
    // find byte within word
    // get first byte into place
    __thisword >>= _S_whichbyte(__prev) * CHAR_BIT;
    for ( size_t __j = _S_whichbyte(__prev); __j < sizeof(_WordT); __j++ ) {
      unsigned char __this_byte
        = static_cast<unsigned char>(__thisword & (~(unsigned char)0));
      if ( __this_byte )
```

```
        return __j*CHAR_BIT + _First_one<true>::_S_first_one[__this_byte];

      __thisword >>= CHAR_BIT;
    }
  }

  // not found, so return a value that indicates failure.
  return __not_found;
} // end _M_do_find_next

//
// One last specialization: _M_do_to_ulong() and the constructor from
// unsigned long are very simple if the bitset consists of a single
// word of type unsigned long.
//

template<>
inline unsigned long
_Base_bitset<1, unsigned long>::_M_do_to_ulong() const { return _M_w; }

template<>
inline _Base_bitset<1, unsigned long>::_Base_bitset(unsigned long __val) {
  _M_w = __val;
}


// ------------------------------------------------------------
// Helper class to zero out the unused high-order bits in the highest word.

template <class _WordT, size_t _Extrabits> struct _Sanitize {
  static void _M_do_sanitize(_WordT& __val)
    { __val &= ~((~static_cast<_WordT>(0)) << _Extrabits); }
};

template <class _WordT> struct _Sanitize<_WordT, 0> {
  static void _M_do_sanitize(_WordT) {}
};

// ------------------------------------------------------------
// Class bitset.
//   _Nb may be any nonzero number of type size_t.
//   Type _WordT may be any unsigned integral type.

template<size_t _Nb, class _WordT = unsigned long>
class bitset : private _Base_bitset<__BITSET_WORDS(_Nb,_WordT), _WordT>
{
private:
  typedef _Base_bitset<__BITSET_WORDS(_Nb,_WordT), _WordT> _Base;
```

```
  // Import base's protected interface.  Necessary because of new template
  // name resolution rules.
  using _Base::_S_whichword;
  using _Base::_S_whichbyte;
  using _Base::_S_whichbit;
  using _Base::_S_maskbit;
  using _Base::_M_getword;
  using _Base::_M_hiword;
  using _Base::_M_do_and;
  using _Base::_M_do_or;
  using _Base::_M_do_xor;
  using _Base::_M_do_left_shift;
  using _Base::_M_do_right_shift;
  using _Base::_M_do_flip;
  using _Base::_M_do_set;
  using _Base::_M_do_reset;
  using _Base::_M_is_equal;
  using _Base::_M_is_any;
  using _Base::_M_do_count;
  using _Base::_M_do_to_ulong;
  using _Base::_M_do_find_first;
  using _Base::_M_do_find_next;

private:
  void _M_do_sanitize() {
    _Sanitize<_WordT,_Nb%__BITS_PER_WORDT(_WordT) >
      ::_M_do_sanitize(_M_hiword());
  }

public:

  // bit reference:
  class reference {
    friend class bitset;

    _WordT *_M_wp;
    size_t _M_bpos;

    // left undefined
    reference();

    reference( bitset& __b, size_t __pos ) {
      _M_wp = &__b._M_getword(__pos);
      _M_bpos = _S_whichbit(__pos);
    }

  public:
    ~reference() {}
```

```
    // for b[i] = __x;
    reference& operator=(bool __x) {
      if ( __x )
        *_M_wp |= _S_maskbit(_M_bpos);
      else
        *_M_wp &= ~_S_maskbit(_M_bpos);

      return *this;
    }

    // for b[i] = b[__j];
    reference& operator=(const reference& __j) {
      if ( (*(__j._M_wp) & _S_maskbit(__j._M_bpos)) )
        *_M_wp |= _S_maskbit(_M_bpos);
      else
        *_M_wp &= ~_S_maskbit(_M_bpos);

      return *this;
    }

    // flips the bit
    bool operator~() const { return (*(_M_wp) & _S_maskbit(_M_bpos)) == 0; }

    // for __x = b[i];
    operator bool() const { return (*(_M_wp) & _S_maskbit(_M_bpos)) != 0; }

    // for b[i].flip();
    reference& flip() {
      *_M_wp ^= _S_maskbit(_M_bpos);
      return *this;
    }
  };

  // 23.3.5.1 constructors:
  bitset() {}
  bitset(unsigned long __val) :
    _Base_bitset<__BITSET_WORDS(_Nb,_WordT), _WordT>(__val) {}

  template<class _CharT, class _Traits, class _Alloc>
  explicit bitset(const basic_string<_CharT,_Traits,_Alloc>& __s,
                  size_t __pos = 0,
                  size_t __n = size_t(basic_string<_CharT,_Traits,_Alloc>::npos))
    : _Base()
  {
    if (__pos > __s.size())
      __STL_THROW(out_of_range("bitset"));
    _M_copy_from_string(__s, __pos, __n);
  }
```

```
// 23.3.5.2 bitset operations:
bitset<_Nb,_WordT>& operator&=(const bitset<_Nb,_WordT>& __rhs) {
  _M_do_and(__rhs);
  return *this;
}

bitset<_Nb,_WordT>& operator|=(const bitset<_Nb,_WordT>& __rhs) {
  _M_do_or(__rhs);
  return *this;
}

bitset<_Nb,_WordT>& operator^=(const bitset<_Nb,_WordT>& __rhs) {
  _M_do_xor(__rhs);
  return *this;
}

bitset<_Nb,_WordT>& operator<<=(size_t __pos) {
  _M_do_left_shift(__pos);
  _M_do_sanitize();
  return *this;
}

bitset<_Nb,_WordT>& operator>>=(size_t __pos) {
  _M_do_right_shift(__pos);
  _M_do_sanitize();
  return *this;
}

//
// Extension:
// Versions of single-bit set, reset, flip, test with no range checking.
//

bitset<_Nb,_WordT>& _Unchecked_set(size_t __pos) {
  _M_getword(__pos) |= _S_maskbit(__pos);
  return *this;
}

bitset<_Nb,_WordT>& _Unchecked_set(size_t __pos, int __val) {
  if (__val)
    _M_getword(__pos) |= _S_maskbit(__pos);
  else
    _M_getword(__pos) &= ~_S_maskbit(__pos);

  return *this;
}

bitset<_Nb,_WordT>& _Unchecked_reset(size_t __pos) {
  _M_getword(__pos) &= ~_S_maskbit(__pos);
```

```
  return *this;
}

bitset<_Nb,_WordT>& _Unchecked_flip(size_t __pos) {
  _M_getword(__pos) ^= _S_maskbit(__pos);
  return *this;
}

bool _Unchecked_test(size_t __pos) const {
  return (_M_getword(__pos) & _S_maskbit(__pos)) != static_cast<_WordT>(0);
}

// Set, reset, and flip.

bitset<_Nb,_WordT>& set() {
  _M_do_set();
  _M_do_sanitize();
  return *this;
}

bitset<_Nb,_WordT>& set(size_t __pos) {
  if (__pos >= _Nb)
    __STL_THROW(out_of_range("bitset"));

  return _Unchecked_set(__pos);
}

bitset<_Nb,_WordT>& set(size_t __pos, int __val) {
  if (__pos >= _Nb)
    __STL_THROW(out_of_range("bitset"));

  return _Unchecked_set(__pos, __val);
}

bitset<_Nb,_WordT>& reset() {
  _M_do_reset();
  return *this;
}

bitset<_Nb,_WordT>& reset(size_t __pos) {
  if (__pos >= _Nb)
    __STL_THROW(out_of_range("bitset"));

  return _Unchecked_reset(__pos);
}

bitset<_Nb,_WordT>& flip() {
  _M_do_flip();
  _M_do_sanitize();
```

```
    return *this;
  }

  bitset<_Nb,_WordT>& flip(size_t __pos) {
    if (__pos >= _Nb)
      __STL_THROW(out_of_range("bitset"));

    return _Unchecked_flip(__pos);
  }

  bitset<_Nb,_WordT> operator~() const {
    return bitset<_Nb,_WordT>(*this).flip();
  }

  // element access:
  //for b[i];
  reference operator[](size_t __pos) { return reference(*this,__pos); }
  bool operator[](size_t __pos) const { return _Unchecked_test(__pos); }

  unsigned long to_ulong() const { return _M_do_to_ulong(); }

#ifdef __STL_EXPLICIT_FUNCTION_TMPL_ARGS
  template <class _CharT, class _Traits, class _Alloc>
  basic_string<_CharT, _Traits, _Alloc> to_string() const {
    basic_string<_CharT, _Traits, _Alloc> __result;
    _M_copy_to_string(__result);
    return __result;
  }
#endif /* __STL_EXPLICIT_FUNCTION_TMPL_ARGS */

  // Helper functions for string operations.
  template<class _CharT, class _Traits, class _Alloc>
  void _M_copy_from_string(const basic_string<_CharT,_Traits,_Alloc>& __s,
                           size_t,
                           size_t);

  // Helper functions for string operations.
  template<class _CharT, class _Traits, class _Alloc>
  void _M_copy_to_string(basic_string<_CharT,_Traits,_Alloc>&) const;

  size_t count() const { return _M_do_count(); }

  size_t size() const { return _Nb; }

  bool operator==(const bitset<_Nb,_WordT>& __rhs) const {
    return _M_is_equal(__rhs);
  }
  bool operator!=(const bitset<_Nb,_WordT>& __rhs) const {
    return !_M_is_equal(__rhs);
```

```
  }

  bool test(size_t __pos) const {
    if (__pos > _Nb)
      __STL_THROW(out_of_range("bitset"));

    return _Unchecked_test(__pos);
  }

  bool any() const { return _M_is_any(); }
  bool none() const { return !_M_is_any(); }

  bitset<_Nb,_WordT> operator<<(size_t __pos) const
    { return bitset<_Nb,_WordT>(*this) <<= __pos; }
  bitset<_Nb,_WordT> operator>>(size_t __pos) const
    { return bitset<_Nb,_WordT>(*this) >>= __pos; }

  //
  // EXTENSIONS: bit-find operations.  These operations are
  // experimental, and are subject to change or removal in future
  // versions.
  //

  // find the index of the first "on" bit
  size_t _Find_first() const
    { return _M_do_find_first(_Nb); }

  // find the index of the next "on" bit after prev
  size_t _Find_next( size_t __prev ) const
    { return _M_do_find_next(__prev, _Nb); }

};

//
// Definitions of non-inline member functions.
//

template <size_t _Nb, class _WordT>
template<class _CharT, class _Traits, class _Alloc>
void bitset<_Nb, _WordT>
  ::_M_copy_from_string(const basic_string<_CharT,_Traits,_Alloc>& __s,
                        size_t __pos,
                        size_t __n)
{
  reset();
  const size_t __nbits = min(_Nb, min(__n, __s.size() - __pos));
  for (size_t __i = 0; __i < __nbits; ++__i) {
    switch(__s[__pos + __nbits - __i - 1]) {
    case '0':
```

```
      break;
    case '1':
      set(__i);
      break;
    default:
      __STL_THROW(invalid_argument("bitset"));
    }
  }
}

template <size_t _Nb, class _WordT>
template <class _CharT, class _Traits, class _Alloc>
void bitset<_Nb, _WordT>
  ::_M_copy_to_string(basic_string<_CharT, _Traits, _Alloc>& __s) const
{
  __s.assign(_Nb, '0');

  for (size_t __i = 0; __i < _Nb; ++__i)
    if (_Unchecked_test(__i))
      __s[_Nb - 1 - __i] = '1';
}

// ------------------------------------------------------------

//
// 23.3.5.3 bitset operations:
//

template <size_t _Nb, class _WordT>
inline bitset<_Nb,_WordT> operator&(const bitset<_Nb,_WordT>& __x,
                                    const bitset<_Nb,_WordT>& __y) {
  bitset<_Nb,_WordT> __result(__x);
  __result &= __y;
  return __result;
}


template <size_t _Nb, class _WordT>
inline bitset<_Nb,_WordT> operator|(const bitset<_Nb,_WordT>& __x,
                                    const bitset<_Nb,_WordT>& __y) {
  bitset<_Nb,_WordT> __result(__x);
  __result |= __y;
  return __result;
}

template <size_t _Nb, class _WordT>
inline bitset<_Nb,_WordT> operator^(const bitset<_Nb,_WordT>& __x,
                                    const bitset<_Nb,_WordT>& __y) {
  bitset<_Nb,_WordT> __result(__x);
```

```
  __result ^= __y;
  return __result;
}

// NOTE: these must be rewritten once we have templatized iostreams.

template <size_t _Nb, class _WordT>
istream&
operator>>(istream& __is, bitset<_Nb,_WordT>& __x) {
  string __tmp;
  __tmp.reserve(_Nb);

  // In new templatized iostreams, use istream::sentry
  if (__is.flags() & ios::skipws) {
    char __c;
    do
      __is.get(__c);
    while (__is && isspace(__c));
    if (__is)
      __is.putback(__c);
  }

  for (size_t __i = 0; __i < _Nb; ++__i) {
    char __c;
    __is.get(__c);

    if (!__is)
      break;
    else if (__c != '0' && __c != '1') {
      __is.putback(__c);
      break;
    }
    else
      __tmp.push_back(__c);
  }

  if (__tmp.empty())
    __is.clear(__is.rdstate() | ios::failbit);
  else
    __x._M_copy_from_string(__tmp, static_cast<size_t>(0), _Nb);

  return __is;
}

template <size_t _Nb, class _WordT>
ostream& operator<<(ostream& __os, const bitset<_Nb,_WordT>& __x) {
  string __tmp;
  __x._M_copy_to_string(__tmp);
  return __os << __tmp;
```

```
}

// ----------------------------------------------------------
// Lookup tables for find and count operations.

template<bool __dummy>
unsigned char _Bit_count<__dummy>::_S_bit_count[] = {
  0, /*   0 */ 1, /*   1 */ 1, /*   2 */ 2, /*   3 */ 1, /*   4 */
  2, /*   5 */ 2, /*   6 */ 3, /*   7 */ 1, /*   8 */ 2, /*   9 */
  2, /*  10 */ 3, /*  11 */ 2, /*  12 */ 3, /*  13 */ 3, /*  14 */
  4, /*  15 */ 1, /*  16 */ 2, /*  17 */ 2, /*  18 */ 3, /*  19 */
  2, /*  20 */ 3, /*  21 */ 3, /*  22 */ 4, /*  23 */ 2, /*  24 */
  3, /*  25 */ 3, /*  26 */ 4, /*  27 */ 3, /*  28 */ 4, /*  29 */
  4, /*  30 */ 5, /*  31 */ 1, /*  32 */ 2, /*  33 */ 2, /*  34 */
  3, /*  35 */ 2, /*  36 */ 3, /*  37 */ 3, /*  38 */ 4, /*  39 */
  2, /*  40 */ 3, /*  41 */ 3, /*  42 */ 4, /*  43 */ 3, /*  44 */
  4, /*  45 */ 4, /*  46 */ 5, /*  47 */ 2, /*  48 */ 3, /*  49 */
  3, /*  50 */ 4, /*  51 */ 3, /*  52 */ 4, /*  53 */ 4, /*  54 */
  5, /*  55 */ 3, /*  56 */ 4, /*  57 */ 4, /*  58 */ 5, /*  59 */
  4, /*  60 */ 5, /*  61 */ 5, /*  62 */ 6, /*  63 */ 1, /*  64 */
  2, /*  65 */ 2, /*  66 */ 3, /*  67 */ 2, /*  68 */ 3, /*  69 */
  3, /*  70 */ 4, /*  71 */ 2, /*  72 */ 3, /*  73 */ 3, /*  74 */
  4, /*  75 */ 3, /*  76 */ 4, /*  77 */ 4, /*  78 */ 5, /*  79 */
  2, /*  80 */ 3, /*  81 */ 3, /*  82 */ 4, /*  83 */ 3, /*  84 */
  4, /*  85 */ 4, /*  86 */ 5, /*  87 */ 3, /*  88 */ 4, /*  89 */
  4, /*  90 */ 5, /*  91 */ 4, /*  92 */ 5, /*  93 */ 5, /*  94 */
  6, /*  95 */ 2, /*  96 */ 3, /*  97 */ 3, /*  98 */ 4, /*  99 */
  3, /* 100 */ 4, /* 101 */ 4, /* 102 */ 5, /* 103 */ 3, /* 104 */
  4, /* 105 */ 4, /* 106 */ 5, /* 107 */ 4, /* 108 */ 5, /* 109 */
  5, /* 110 */ 6, /* 111 */ 3, /* 112 */ 4, /* 113 */ 4, /* 114 */
  5, /* 115 */ 4, /* 116 */ 5, /* 117 */ 5, /* 118 */ 6, /* 119 */
  4, /* 120 */ 5, /* 121 */ 5, /* 122 */ 6, /* 123 */ 5, /* 124 */
  6, /* 125 */ 6, /* 126 */ 7, /* 127 */ 1, /* 128 */ 2, /* 129 */
  2, /* 130 */ 3, /* 131 */ 2, /* 132 */ 3, /* 133 */ 3, /* 134 */
  4, /* 135 */ 2, /* 136 */ 3, /* 137 */ 3, /* 138 */ 4, /* 139 */
  3, /* 140 */ 4, /* 141 */ 4, /* 142 */ 5, /* 143 */ 2, /* 144 */
  3, /* 145 */ 3, /* 146 */ 4, /* 147 */ 3, /* 148 */ 4, /* 149 */
  4, /* 150 */ 5, /* 151 */ 3, /* 152 */ 4, /* 153 */ 4, /* 154 */
  5, /* 155 */ 4, /* 156 */ 5, /* 157 */ 5, /* 158 */ 6, /* 159 */
  2, /* 160 */ 3, /* 161 */ 3, /* 162 */ 4, /* 163 */ 3, /* 164 */
  4, /* 165 */ 4, /* 166 */ 5, /* 167 */ 3, /* 168 */ 4, /* 169 */
  4, /* 170 */ 5, /* 171 */ 4, /* 172 */ 5, /* 173 */ 5, /* 174 */
  6, /* 175 */ 3, /* 176 */ 4, /* 177 */ 4, /* 178 */ 5, /* 179 */
  4, /* 180 */ 5, /* 181 */ 5, /* 182 */ 6, /* 183 */ 4, /* 184 */
  5, /* 185 */ 5, /* 186 */ 6, /* 187 */ 5, /* 188 */ 6, /* 189 */
  6, /* 190 */ 7, /* 191 */ 2, /* 192 */ 3, /* 193 */ 3, /* 194 */
  4, /* 195 */ 3, /* 196 */ 4, /* 197 */ 4, /* 198 */ 5, /* 199 */
  3, /* 200 */ 4, /* 201 */ 4, /* 202 */ 5, /* 203 */ 4, /* 204 */
  5, /* 205 */ 5, /* 206 */ 6, /* 207 */ 3, /* 208 */ 4, /* 209 */
```

```
  4, /* 210 */ 5, /* 211 */ 4, /* 212 */ 5, /* 213 */ 5, /* 214 */
  6, /* 215 */ 4, /* 216 */ 5, /* 217 */ 5, /* 218 */ 6, /* 219 */
  5, /* 220 */ 6, /* 221 */ 6, /* 222 */ 7, /* 223 */ 3, /* 224 */
  4, /* 225 */ 4, /* 226 */ 5, /* 227 */ 4, /* 228 */ 5, /* 229 */
  5, /* 230 */ 6, /* 231 */ 4, /* 232 */ 5, /* 233 */ 5, /* 234 */
  6, /* 235 */ 5, /* 236 */ 6, /* 237 */ 6, /* 238 */ 7, /* 239 */
  4, /* 240 */ 5, /* 241 */ 5, /* 242 */ 6, /* 243 */ 5, /* 244 */
  6, /* 245 */ 6, /* 246 */ 7, /* 247 */ 5, /* 248 */ 6, /* 249 */
  6, /* 250 */ 7, /* 251 */ 6, /* 252 */ 7, /* 253 */ 7, /* 254 */
  8  /* 255 */
}; // end _Bit_count

template<bool __dummy>
unsigned char _First_one<__dummy>::_S_first_one[] = {
  0, /*   0 */ 0, /*   1 */ 1, /*   2 */ 0, /*   3 */ 2, /*   4 */
  0, /*   5 */ 1, /*   6 */ 0, /*   7 */ 3, /*   8 */ 0, /*   9 */
  1, /*  10 */ 0, /*  11 */ 2, /*  12 */ 0, /*  13 */ 1, /*  14 */
  0, /*  15 */ 4, /*  16 */ 0, /*  17 */ 1, /*  18 */ 0, /*  19 */
  2, /*  20 */ 0, /*  21 */ 1, /*  22 */ 0, /*  23 */ 3, /*  24 */
  0, /*  25 */ 1, /*  26 */ 0, /*  27 */ 2, /*  28 */ 0, /*  29 */
  1, /*  30 */ 0, /*  31 */ 5, /*  32 */ 0, /*  33 */ 1, /*  34 */
  0, /*  35 */ 2, /*  36 */ 0, /*  37 */ 1, /*  38 */ 0, /*  39 */
  3, /*  40 */ 0, /*  41 */ 1, /*  42 */ 0, /*  43 */ 2, /*  44 */
  0, /*  45 */ 1, /*  46 */ 0, /*  47 */ 4, /*  48 */ 0, /*  49 */
  1, /*  50 */ 0, /*  51 */ 2, /*  52 */ 0, /*  53 */ 1, /*  54 */
  0, /*  55 */ 3, /*  56 */ 0, /*  57 */ 1, /*  58 */ 0, /*  59 */
  2, /*  60 */ 0, /*  61 */ 1, /*  62 */ 0, /*  63 */ 6, /*  64 */
  0, /*  65 */ 1, /*  66 */ 0, /*  67 */ 2, /*  68 */ 0, /*  69 */
  1, /*  70 */ 0, /*  71 */ 3, /*  72 */ 0, /*  73 */ 1, /*  74 */
  0, /*  75 */ 2, /*  76 */ 0, /*  77 */ 1, /*  78 */ 0, /*  79 */
  4, /*  80 */ 0, /*  81 */ 1, /*  82 */ 0, /*  83 */ 2, /*  84 */
  0, /*  85 */ 1, /*  86 */ 0, /*  87 */ 3, /*  88 */ 0, /*  89 */
  1, /*  90 */ 0, /*  91 */ 2, /*  92 */ 0, /*  93 */ 1, /*  94 */
  0, /*  95 */ 5, /*  96 */ 0, /*  97 */ 1, /*  98 */ 0, /*  99 */
  2, /* 100 */ 0, /* 101 */ 1, /* 102 */ 0, /* 103 */ 3, /* 104 */
  0, /* 105 */ 1, /* 106 */ 0, /* 107 */ 2, /* 108 */ 0, /* 109 */
  1, /* 110 */ 0, /* 111 */ 4, /* 112 */ 0, /* 113 */ 1, /* 114 */
  0, /* 115 */ 2, /* 116 */ 0, /* 117 */ 1, /* 118 */ 0, /* 119 */
  3, /* 120 */ 0, /* 121 */ 1, /* 122 */ 0, /* 123 */ 2, /* 124 */
  0, /* 125 */ 1, /* 126 */ 0, /* 127 */ 7, /* 128 */ 0, /* 129 */
  1, /* 130 */ 0, /* 131 */ 2, /* 132 */ 0, /* 133 */ 1, /* 134 */
  0, /* 135 */ 3, /* 136 */ 0, /* 137 */ 1, /* 138 */ 0, /* 139 */
  2, /* 140 */ 0, /* 141 */ 1, /* 142 */ 0, /* 143 */ 4, /* 144 */
  0, /* 145 */ 1, /* 146 */ 0, /* 147 */ 2, /* 148 */ 0, /* 149 */
  1, /* 150 */ 0, /* 151 */ 3, /* 152 */ 0, /* 153 */ 1, /* 154 */
  0, /* 155 */ 2, /* 156 */ 0, /* 157 */ 1, /* 158 */ 0, /* 159 */
  5, /* 160 */ 0, /* 161 */ 1, /* 162 */ 0, /* 163 */ 2, /* 164 */
  0, /* 165 */ 1, /* 166 */ 0, /* 167 */ 3, /* 168 */ 0, /* 169 */
  1, /* 170 */ 0, /* 171 */ 2, /* 172 */ 0, /* 173 */ 1, /* 174 */
```

```
  0, /* 175 */ 4, /* 176 */ 0, /* 177 */ 1, /* 178 */ 0, /* 179 */
  2, /* 180 */ 0, /* 181 */ 1, /* 182 */ 0, /* 183 */ 3, /* 184 */
  0, /* 185 */ 1, /* 186 */ 0, /* 187 */ 2, /* 188 */ 0, /* 189 */
  1, /* 190 */ 0, /* 191 */ 6, /* 192 */ 0, /* 193 */ 1, /* 194 */
  0, /* 195 */ 2, /* 196 */ 0, /* 197 */ 1, /* 198 */ 0, /* 199 */
  3, /* 200 */ 0, /* 201 */ 1, /* 202 */ 0, /* 203 */ 2, /* 204 */
  0, /* 205 */ 1, /* 206 */ 0, /* 207 */ 4, /* 208 */ 0, /* 209 */
  1, /* 210 */ 0, /* 211 */ 2, /* 212 */ 0, /* 213 */ 1, /* 214 */
  0, /* 215 */ 3, /* 216 */ 0, /* 217 */ 1, /* 218 */ 0, /* 219 */
  2, /* 220 */ 0, /* 221 */ 1, /* 222 */ 0, /* 223 */ 5, /* 224 */
  0, /* 225 */ 1, /* 226 */ 0, /* 227 */ 2, /* 228 */ 0, /* 229 */
  1, /* 230 */ 0, /* 231 */ 3, /* 232 */ 0, /* 233 */ 1, /* 234 */
  0, /* 235 */ 2, /* 236 */ 0, /* 237 */ 1, /* 238 */ 0, /* 239 */
  4, /* 240 */ 0, /* 241 */ 1, /* 242 */ 0, /* 243 */ 2, /* 244 */
  0, /* 245 */ 1, /* 246 */ 0, /* 247 */ 3, /* 248 */ 0, /* 249 */
  1, /* 250 */ 0, /* 251 */ 2, /* 252 */ 0, /* 253 */ 1, /* 254 */
  0, /* 255 */
}; // end _First_one

#if defined(__sgi) && !defined(__GNUC__) && (_MIPS_SIM != _MIPS_SIM_ABI32)
#pragma reset woff 1209
#endif

__STL_END_NAMESPACE


#undef __BITS_PER_WORDT
#undef __BITSET_WORDS

#endif /* __SGI_STL_BITSET */


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