Doc No: SC22/WG21/N2050
J16/06-0120
Date: 2006-06-26
Project: JTC1.22.32
Reply to: Jamie Allsop <ja11sop@yahoo.co.uk>
Alisdair Meredith <Alisdair.Meredith@uk.renaultf1.com>
Gennaro Prota <gennaro_prota@yahoo.com>

Proposal to Add a Dynamically Sizeable Bitset to the Standard Library Technical Report

Revision 1

I Motivation and Introduction

Manipulating a set of bits is an often desired task in C++. Typically the set of bits is representative of a set of flags or is used as a mask where each bit represents a domain specific entity, such as a day of a year. Typically a more complex mask will be composed by concatenating smaller masks, such as a masks representing valid days in a given month being concatenated into a mask representing a whole year. A set of bits is also (not surprisingly) useful for performing set operations.

A container of `bool`s can be utilised to represent such sets of bits or masks but often there is a desire to manipulate a set of bits rather than a container of `bool`s, typically for efficiency or space reasons. Efficiency is gained through the use of the bitwise and shift operators, and space can be minimised if only one bit is used to store each value in the set. By providing such bitwise and shift operators and using a packed representation it is possible to provide an efficient represention of arbitrary length binary numbers. This is also one of the features of this proposal.

The standard library currently provides two libraries which can be used to represent a set of bits, `std::vector<bool>` and `std:bitset`.

The problems of `std::vector<bool>` are well documented in [N1185 – Sutter, 1999], [N1847 – Sutter, 1999] and [Meyers, 2001]. Notably, `std::vector<bool>` is not a container and `std::vector<bool>::iterator` does not meet the requirements of a Random access iterator, though [N1640 – Abrahams, Siek, Witt, 2004] may see a solution to this problem. While the interface of `std::vector<bool>` does support a packed representation it does not provide the bitwise and shift operators. Further the LWG has expressed a desire to deprecate this specialization, and provide a new class specifically designed for these optimizations.

On the other hand `std::bitset` does provide bitwise and shift operators and uses a packed bit representation though the size of `std::bitset` is fixed at compile time. Obviously then the possibility of resizing a `std::bitset` or composing a new `std::bitset` by concatenating two or more `std::bitset`s does not exist.

Of course it is entirely possible to implement a dynamically sized set of bits as and when one is required but this is a non-trivial exercise, as noted in [Sutter, 2005]. Therefore this paper proposes a dynamically sizeable

bitset library similar in intention to `std::bitset`. Previously [N0220 – Allison, 1993] proposed a `bitstring` class which was dynamically sizeable and provided bitwise and shift operations. More recently such a library was accepted into Boost [Boost Libraries] on the 18th of June 2002, called `boost::dynamic_bitset` [Boost – dynamic_bitset], previously known as `dyn_bitset`. Searching for 'bitset' in the subject title in the Boost developer's mailing list [Gmane – boost.devel] using a newsreader will reveal most of the discussions that have occurred in relation to this library. The proposal here is based on the `boost::dynamic_bitset` library, with differences noted in the paper.

II Impact On the Standard

This proposal is a pure library extension. It does not require changes to any standard classes or functions and it does not require changes to any of the standard requirement tables. It does not require any changes in the core language, and it has been implemented in standard C++. The proposal does not depend on any other library extensions.

III Design Decisions

1 Introduction

The `dynamic_bitset` class outlined in this paper represents a sequence of bits. It provides access to the value of individual bits via `operator[]` and provides all of the bitwise operators, such as `operator&` and the shift operators, such as `operator<<`. The number of bits in the set is specified at runtime via a parameter to the constructor of the `dynamic_bitset`. The proposal here closely follows the `boost::dynamic_bitset` library with some deviations, for example the addition of an `append()` member function to allow one `dynamic_bitset` to be appended to another `dynamic_bitset`.

The `dynamic_bitset` class interface is closely modelled on the `std::bitset` interface and the name is chosen deliberately to imply this relationship. Alternative names might have been `bitvector` or `bitstring` but `dynamic_bitset` was chosen as it best captures the intention of the class. The primary difference between `std::bitset` and `dynamic_bitset` is the ability to modify the bitset size at runtime. As the interface of `dynamic_bitset` has been modelled closely on `std::bitset` choice of member function names have followed the existing practice of `std::bitset` such as using 'flip' instead of 'toggle'.

The `dynamic_bitset` class is designed to solve two key problems. First it can be used to represent a subset of a finite set, where each bit represents whether an element of the finite set is in the subset or not. Second, it can used to represent an arbitrary sized binary number. As such the bitwise operations of `dynamic_bitset`, such as `operator&` and `operator|`, are provided and correspond to the set operations, intersection and union respectively. In addition, set difference is also supported using `operator-`. Similarly `dynamic_bitset` also provides the shift operators.

Finally here are some definitions for terms that are used in the remainder of this document. Each bit represents either the Boolean value `true` or `false` (`1` or `0`). To *set* a bit is to assign it `1`. To *clear* or *reset* a bit is to assign it `0`. To *flip* (or *toggle*) a bit is to change the value to `1` if it was `0` and to `0` if it was `1`. Each bit has a non-negative position. A `dynamic_bitset` *x* contains *x*`.size()` bits, with each bit assigned a unique position in the range `[0,`*x*`.size())`. The bit at position `0` is called the *least significant bit* and the bit at position `size() - 1` is the *most significant bit*. When converting an instance of `dynamic_bitset` to, or from, an `unsigned long long` *u*, the bit at position *i* of the `dynamic_bitset` has the same value as `(`*u* `>>` *i*`) & 1`.

2 Constructing `dynamic_bitset`s using strings

By default when constructing a `dynamic_bitset` using a string it is considered that the first character in

the string is the most significant bit, in other words the string is considered to be representative of a binary number. This bit ordering assumption is controlled by the `bit_order` argument to the constructor, which by default is `msb_first`, but can be changed to `lsb_first`. Therefore `dynamic_bitset(std::string("10001110")) == dynamic_bitset(4, 142ull)`. Also given `dynamic_bitset<> dbs1(std::string("10001110"));` we have `dbs1[0] == 0` and `dbs1[7] == 1`. Given `dynamic_bitset<> dbs2(std::string("10001110"),lsb_first);` we have `dbs2[0] == 1` and `dbs2[7] == 0`.

The ability to specify bit ordering of the string allows for compatibility with existing code, no matter which bit-order representation was chosen.

3 Conversion to string and streaming

Like `std::bitset` a `to_string()` function is provided, but unlike `std::bitset` this is not in the form of overloaded member functions returning `std::basic_string<CharT, traits, Allocator>`, rather as a `void` non-member function which modifies a `StringT` passed by reference. This simplifies the use of the function by not requiring the user to explicitly specify the string's template parameters, for example if given `std::bitset` *x* we might call `to_string()` as,

x`.template to_string<charT,traits,allocator<charT> >().`

Another feature of `to_string()` is the ability to specify the required bit order of the result. Again this is specified through the use of the `bit_order` enum. By default this is `msb_first`. Similarly the default bit order used when writing the `dynamic_bitset` to a stream is also to write the most significant bit first. It is possible to specify the bit order by using the `set_bit_order()` manipulator passing the required `bit_order` as an argument.

4 Confusion between streaming and shifting operators

Both streaming and shift operators are provided. It may at first appear that this would lead to confusion. However in practice this is unlikely to occur. For example, given,

```
dynamic_bitset<> dbs(std::string("11111111"));
```

we have,

```
std::cout << dbs << std::endl;           // ==  11111111
std::cout << dbs << 3 << std::endl;      // ==  111111113
std::cout << (dbs << 3) << std::endl;    // ==  11111000
std::cout << (dbs >> 3) << std::endl;    // ==  00011111
```

This is quite straightforward and what would be expected.

5 Concatenating `dynamic_bitset`s

The `append()` member function supports appending `dynamic_bitset`s together. It would seem to make sense to extend the interface of `dynamic_bitset` to allow the concatenation of `dynamic_bitset`s using `operator+=` and `operator+`, however this may be confusing as `operator-=` is already in use to calculate the set difference. Therefore these operators have not been provided for this proposal, though if there was consensus that this would be a good thing then there is no reason not to look at this again.

6 Using `replace(size_type` *pos*`, bool` *val*`)` instead of `set(size_type` *pos*`, bool` *val* `= true)`

This proposal suggests using `replace(size_type` *pos*`, bool` *val*`)` and set(size_type pos) instead of `set(size_type` *pos*`, bool` *val* `= true)` as `std::bitset` does. The reason for this is to reinforce the notion that to 'set' a bit in the bitset is to set it to 1. If there is a consensus that it is important to maintain the

same interface as `std::bitset` then the `std::bitset` interface could be used instead.

One advantage of the proposed interface here is that the `find_*_set()` and `find_*_reset()` functions are not ambiguous.

7 Bitwise operators do not require `dynamic_bitset`s of the same size

Operators work as expected up to the largest valid index in both `dynamic_bitset`s and then treats non-invalid indexes in the smaller of the two `dynamic_bitset`s as `0`s.

8 Lookup using `find_*_set()` and `find_*_reset()`

Although iterators are not provided the following lookup functions allow traversal of the bitset,

```
size_type find_first_set() const;
size_type find_next_set(size_type pos) const;
size_type find_last_set() const;
size_type find_prev_set(size_type pos) const;
size_type find_first_reset() const;
size_type find_next_reset(size_type pos) const;
size_type find_last_reset() const;
size_type find_prev_reset(size_type pos) const;
```

A possible alternative naming convention would be to use the word `clear` instead of `reset`, though this would cause a disparity with, the `set()` and `reset()` functions.

Another alternative would to use a style similar to that used in `std::basic_string`, either this,

```
size_type find_first() const;
size_type find_next(size_type pos) const;
size_type find_last() const;
size_type find_prev(size_type pos) const;
size_type find_first_not() const;
size_type find_next_not(size_type pos) const;
size_type find_last_not() const;
size_type find_prev_not(size_type pos) const;
```

or,

```
size_type find_first_of(bool bit) const;
size_type find_next_of(bool bit , size_type pos) const;
size_type find_last_of(bool bit ) const;
size_type find_prev_of(bool bit , size_type pos) const;
```

If `set(size_type pos, bool val = true)` is favoured over `replace(size_type pos, bool val)` then this naming convention may be more appropriate.

IV Proposed Text

1 Header <dynamic_bitset> synopsis

```
namespace std {
enum bit_order { lsb_first, msb_first };

template <typename Block = unsigned long long,
          typename Allocator = std::allocator<Block> >
class dynamic_bitset;

// swap:
template <typename Block,
          typename Allocator>
void swap(dynamic_bitset<Block, Allocator>& lhs,
          dynamic_bitset<Block, Allocator>& rhs);

// dynamic_bitset operations:
template <typename Block,
          typename Allocator>
bool operator==(const dynamic_bitset<Block, Allocator>& lhs,
                const dynamic_bitset<Block, Allocator>& rhs);

template <typename Block,
          typename Allocator>
bool operator!=(const dynamic_bitset<Block, Allocator>& lhs,
                const dynamic_bitset<Block, Allocator>& rhs);

template <typename Block, typename Allocator>
bool operator<(const dynamic_bitset<Block, Allocator>& lhs,
               const dynamic_bitset<Block, Allocator>& rhs);

template <typename Block,
          typename Allocator>
bool operator<=(const dynamic_bitset<Block, Allocator>& lhs,
                const dynamic_bitset<Block, Allocator>& rhs);

template <typename Block,
          typename Allocator>
bool operator>(const dynamic_bitset<Block, Allocator>& lhs,
               const dynamic_bitset<Block, Allocator>& rhs);

template <typename Block,
          typename Allocator>
bool operator>=(const dynamic_bitset<Block, Allocator>& lhs,
                const dynamic_bitset<Block, Allocator>& rhs);

template <typename Block,
          typename Allocator>
dynamic_bitset<Block, Allocator>
operator&(const dynamic_bitset<Block, Allocator>& lhs,
          const dynamic_bitset<Block, Allocator>& rhs);
```

```
template <typename Block,
          typename Allocator>
dynamic_bitset<Block, Allocator>
operator|(const dynamic_bitset<Block, Allocator>& lhs,
          const dynamic_bitset<Block, Allocator>& rhs);

template <typename Block,
          typename Allocator>
dynamic_bitset<Block, Allocator>
operator^(const dynamic_bitset<Block, Allocator>& lhs,
          const dynamic_bitset<Block, Allocator>& rhs);

template <typename Block,
          typename Allocator>
dynamic_bitset<Block, Allocator>
operator-(const dynamic_bitset<Block, Allocator>& lhs,
          const dynamic_bitset<Block, Allocator>& rhs);

template <typename Block,
          typename Allocator>
bool intersect(const dynamic_bitset<Block, Allocator>& lhs,
               const dynamic_bitset<Block, Allocator>& rhs,
               int offset = 0);

template <typename Block,
          typename Allocator>
bool disjoint(const dynamic_bitset<Block, Allocator>& lhs,
              const dynamic_bitset<Block, Allocator>& rhs,
              int offset = 0);

template <typename Block,
          typename Allocator,
          typename StringT>
void to_string(const dynamic_bitset<Block, Allocator>& x,
               StringT& str,
               bit_order order = msb_first);

template <typename Block,
          typename Allocator,
          typename BlockOutputIterator>
void to_block_range(const dynamic_bitset<Block, Allocator>& x,
                    BlockOutputIterator result);

template <typename CharT,
          typename Traits,
          typename Block,
          typename Allocator>
std::basic_ostream<CharT, Traits>&
```

```
operator<<(std::basic_ostream<CharT, Traits>& os,
           const dynamic_bitset<Block, Allocator>& x);

template <typename CharT,
          typename Traits,
          typename Block,
          typename Allocator>
std::basic_istream<CharT, Traits>&
operator>>(std::basic_istream<CharT, Traits>& is,
           dynamic_bitset<Block, Allocator>& x);

// dynamic_bitset bit_order stream manipulators:
class set_bit_order {
public:
  explicit
  set_bit_order(bit_order o);
};

template<class CharT, class Traits>
std::basic_ostream<CharT, Traits>& operator<<
  (std::basic_ostream<CharT, Traits>& os, const set_bit_order& sbo);

} // namespace std
```

The header <dynamic_bitset> defines a class template and several related functions for representing and manipulating a dynamically sized sequences of bits.

```
namespace std {

  template <typename Block,
            typename Allocator>
  class dynamic_bitset {
  public:
    typedef Block                   block_type;
    typedef Allocator               allocator_type;
    typedef implementation-defined  size_type;
    typedef implementation-defined  block_width_type;

    static const block_width_type bits_per_block = implementation-defined;
    static const size_type        npos           = implementation-defined;

    // bit reference:
    class reference
    {
        friend class dynamic_bitset<Block, Allocator>;

        reference(block_type& x, int pos);

        void operator&(); // not defined

    public:
```

```
    reference& operator=(bool x);                  // for b[i] = x;
    reference& operator=(const reference& );       // for b[i] = b[j];

    reference& operator|=(bool x);                 // for b[i] = b[i] | x;
    reference& operator&=(bool x);                 // for b[i] = b[i] & x;
    reference& operator^=(bool x);                 // for b[i] = b[i] ^ x;
    reference& operator-=(bool x);                 // for b[i] = b[i] & !x;

    bool operator~() const;                        // flips the bits
    operator bool() const;                         // for x = b[i]
    reference& flip();                             // for b[i].flip();
};

typedef bool const_reference;

// constructors:
explicit
dynamic_bitset(const Allocator& alloc = Allocator());

explicit
dynamic_bitset(size_type num_bits,
               unsigned long long value = 0,
               const Allocator& alloc = Allocator());

template <typename BlockInputIterator>
dynamic_bitset(BlockInputIterator first,
               BlockInputIterator last,
               const Allocator& alloc = Allocator());

template <typename CharT,
          typename Traits,
          typename Alloc>
explicit
dynamic_bitset(
  const std::basic_string<CharT, Traits, Alloc>& str,
  bit_order order = msb_first,
  typename std::basic_string<CharT, Traits, Alloc>::size_type pos = 0,
  typename std::basic_string<CharT, Traits, Alloc>::size_type n =
    std::basic_string<CharT, Traits, Alloc>::npos,
  size_type num_bits = npos,
  const Allocator& alloc = Allocator());

dynamic_bitset(const dynamic_bitset& x);

dynamic_bitset(const dynamic_bitset& x,
               size_type pos,
               size_type n = npos,
               size_type num_bits = npos,
```

```
                const Allocator& alloc = Allocator());

  // destructor:
  ~dynamic_bitset();

  // swap:
  void swap(dynamic_bitset& x);

  // assignment:
  dynamic_bitset& operator=(const dynamic_bitset& x);

  // allocator:
  allocator_type get_allocator() const;

  // modifiers:
  void resize(size_type num_bits, bool value = false);
  void clear();
  void push_back(bool bit);
  void append(Block block);
  template <typename BlockInputIterator>
  void append(BlockInputIterator first, BlockInputIterator last);
  void append(const dynamic_bitset& x);
  void assign(const dynamic_bitset& x,
              size_type pos,
              size_type n = npos,
              size_type num_bits = npos);

  // bitwise operations:
  dynamic_bitset& operator&=(const dynamic_bitset& rhs);
  dynamic_bitset& operator|=(const dynamic_bitset& rhs);
  dynamic_bitset& operator^=(const dynamic_bitset& rhs);
  dynamic_bitset& operator-=(const dynamic_bitset& rhs);

  // bit shift operations:
  dynamic_bitset& operator<<=(size_type n);
  dynamic_bitset& operator>>=(size_type n);
  dynamic_bitset  operator<<(size_type n) const;
  dynamic_bitset  operator>>(size_type n) const;

  // basic bit operations:
  dynamic_bitset& replace(size_type pos, bool val);
  dynamic_bitset& set(size_type pos);
  dynamic_bitset& set();
  dynamic_bitset& reset(size_type pos);
  dynamic_bitset& reset();
  dynamic_bitset& flip(size_type pos);
  dynamic_bitset& flip();
```

```
    dynamic_bitset  operator~() const;
    bool test(size_type pos) const;
    bool any() const;
    bool none() const;
    bool all() const;
    size_type count() const;

    // element access:
    reference operator[](size_type pos);
    bool operator[](size_type pos) const;

    // conversion:
    unsigned long long to_ulonglong() const;

    // capacity:
    size_type size() const;
    size_type num_blocks() const;
    size_type max_size() const;
    bool empty() const;

    // set queries:
    bool is_subset_of(const dynamic_bitset& x,
                      int offset = 0,
                      size_type n = npos) const;
    bool is_proper_subset_of(const dynamic_bitset& x,
                             int offset = 0,
                             size_type n = npos) const;
    bool is_superset_of(const dynamic_bitset& x,
                        int offset = 0,
                        size_type n = npos) const;
    bool is_proper_superset_of(const dynamic_bitset& x,
                               int offset = 0,
                               size_type n = npos) const;

    // lookup:
    size_type find_first_set() const;
    size_type find_next_set(size_type pos) const;
    size_type find_last_set() const;
    size_type find_prev_set(size_type pos) const;
    size_type find_first_reset() const;
    size_type find_next_reset(size_type pos) const;
    size_type find_last_reset() const;
    size_type find_prev_reset(size_type pos) const;

};

} // namespace std
```

2 `dynamic_bitset` constructors

```
explicit
dynamic_bitset(const Allocator& alloc = Allocator());
```

Effects: Constructs a `dynamic_bitset` of size zero. A copy of the *alloc* object will be used in subsequent `dynamic_bitset` operations such as `resize` to allocate memory.

Postconditions: `this->size() == 0`.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

```
explicit
dynamic_bitset(size_type num_bits,
               unsigned long long value = 0,
               const Allocator& alloc = Allocator());
```

Effects: Constructs a `dynamic_bitset` from an `unsigned long long`. The first *M* bits are initialized to the corresponding bits in *value* and all other bits, if any, to zero (where *M* = `min(`*num_bits*`, std::numeric_limits<unsigned long long>::digits))`. A copy of the *alloc* object will be used in subsequent `dynamic_bitset` operations such as `resize` to allocate memory.

Postconditions:

- `this->size() ==` *num_bits*.
- For all *i* in the range `[0,`*M*`), (*this)[`*i*`] == (`*value* `>>` *i*`) & 1`.
- For all *i* in the range `[`*M*`,`*num_bits*`), (*this)[`*i*`] == false`.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

```
template <typename CharT,
          typename Traits,
          typename Alloc>
explicit
dynamic_bitset(
  const std::basic_string<CharT, Traits, Alloc>& str,
  bit_order order = msb_first,
  typename std::basic_string<CharT, Traits, Alloc>::size_type pos = 0,
  typename std::basic_string<CharT, Traits, Alloc>::size_type n =
    std::basic_string<CharT, Traits, Alloc>::npos,
  size_type num_bits = npos,
  const Allocator& alloc = Allocator());
```

Requires: pos `<=` *str*`.size()` and the characters used to initialize the bits must be `0` or `1`.

Effects: Constructs a `dynamic_bitset` from a string of `0`'s and `1`'s. The first *M* bits are initialized to the corresponding characters in *str*, where *M* = `min(min(`*str*`.size() -` *pos*`,` *n*`),` *num_bits*`)`, if `n != std::basic_string<CharT, Traits, Alloc>::npos` and *num_bits* `!= npos`.

If *n* `== std::basic_string<CharT, Traits, Alloc>::npos` then *n* is ignored. Similarly, if *num_bits* `== npos` it is also ignored.

If *order* is `msb_first` then the highest character position in *str* (the rightmost character), not the

lowest (leftmost character), corresponds to the least significant bit. That is, character position *pos* + *M* - 1 - *i* corresponds to bit *i*. Otherwise, if *order* is `lsb_first` then character position *pos* corresponds to bit *i*.

[*Example:* `dynamic_bitset(string("11011100"))` is the same as `dynamic_bitset(220ul)`. —*end example*].

Throws: an allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

```
template <typename BlockInputIterator>
dynamic_bitset(BlockInputIterator first,
               BlockInputIterator last,
               const Allocator& alloc = Allocator());
```

Requires: The type `BlockInputIterator` must be a model of Input Iterator and its `value_type` must be the same type as `Block`.

Effects: Constructs a `dynamic_bitset` based on a range of blocks. Let `*`*first* be block number `0`, `*++`*first* block number `1`, etc. Block number *b* is used to initialize the bits of the `dynamic_bitset` in the position range `[`*b*`*bits_per_block, (`*b*`+1)*bits_per_block)`. For each block number *b* with value *blockvalue*, the bit `(` *blockvalue* `>>` *i*`) & 1` corresponds to the bit at position `(`*b* `* bits_per_block +` *i*`)` in the `dynamic_bitset` (where *i* goes through the range `[0, bits_per_block)`).

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

```
dynamic_bitset(const dynamic_bitset& x,
               size_type pos,
               size_type n = npos,
               size_type num_bits = npos,
               const Allocator& alloc = Allocator());
```

Requires: pos is a valid index in *x*.

Effects: Constructs a `dynamic_bitset` that is a subset of the `dynamic_bitset` *x*. The first *M* bits are initialized to bits `[`*pos*`,`*pos*`+`*M*`)` in *x*, where *M* `= min(`*n*`, `*x*`.size()-`*pos*`, `*num_bits*`)`. If *n* is `npos` then *n* is considered to be *x*`.size()-`*pos*. If *num_bits* is `npos` then *num_bits* is considered to be `min(`*n*`, `*x*`.size()-`*pos*`)`. The remaining bits, up to *num_bits*`-1` are initialized to `0`.

Postconditions:

- `this->size()` == *num_bits*.
- For all *i* in the range `[0,`*M*`)`, `(*this)[`*i*`]` == *x*`[`*i*`+`*pos*`]`.
- For all *i* in the range `[`*M*`,`*num_bits*`)`, `(*this)[`*i*`] == false`.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

```
dynamic_bitset(const dynamic_bitset& x);
```

Effects: Constructs a `dynamic_bitset` that is a copy of the `dynamic_bitset` *x*. The allocator for this `dynamic_bitset` is a copy of the allocator in *x*.

Postconditions: For all *i* in the range `[0,x.size()), (*this)[i] == x[i]`.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

3 `dynamic_bitset` swap

```
void swap(dynamic_bitset& x);
```

Effects: The contents of this `dynamic_bitset` and `dynamic_bitset` *x* are exchanged.

Postconditions: This `dynamic_bitset` is equal to the original *x*, and *x* is equal to the previous version of this `dynamic_bitset`.

Throws: nothing.

4 `dynamic_bitset` assignment

```
dynamic_bitset& operator=(const dynamic_bitset& x);
```

Effects: This `dynamic_bitset` becomes a copy of the `dynamic_bitset` *x*.

Postconditions: For all *i* in the range `[0,x.size()), (*this)[i] == x[i]`.

Returns: `*this`.

Throws: nothing.

5 `dynamic_bitset` allocator

```
allocator_type get_allocator() const;
```

Returns: A copy of the allocator object used to construct `*this`.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

6 `dynamic_bitset` resizing

```
void resize(size_type num_bits, bool value = false);
```

Effects: Changes the number of bits of the dynamic_bitset to *num_bits*. If *num_bits* `> size()` then the bits in the range `[0,size())` remain the same, and the bits in `[size(),`*num_bits*`)` are all set to *value*. If *num_bits* `< size()` then the bits in the range `[0,`*num_bits*`)` stay the same and the remaining bits are discarded.

Postconditions: `this->size() ==` *num_bits*.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

```
void clear();
```

Effects: The size of the `dynamic_bitset` becomes zero.

Throws: nothing.

`void push_back(bool` *bit*`);`

Effects: Increases the size of the `dynamic_bitset` by one, and sets the value of the new most-significant bit to *bit*.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

`void append(Block` *block*`);`

Effects: Appends the bits in value to the `dynamic_bitset` (appends to the most-significant end). This increases the size of the `dynamic_bitset` by `bits_per_block`. Let *s* be the old size of the `dynamic_bitset`, then for *i* in the range `[0,bits_per_block)`, the bit at position `(`*s* `+` *i*`)` is set to `((`*block* `>>` *i*`) & 1)`.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

```
template <typename BlockInputIterator>
void append(BlockInputIterator first, BlockInputIterator last);
```

Requires: The `BlockInputIterator` type must be a model of Input Iterator and the `value_type` must be the same type as `Block`.

Effects: The result is equivalent to:

```
for(; first != last; ++first)
  this->append(*first);
```

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

`void append(const dynamic_bitset&` *x*`);`

Effects: The result is equivalent to:

```
for(size_type i=0; i<x.size(); ++i)
  this->push_back(x[i]);
```

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

```
void assign(const dynamic_bitset& x,
            size_type pos,
            size_type n = npos,
            size_type num_bits = npos);
```

Requires: pos is a valid index in *x*.

Effects: Equivalent to the assignment of a newly constructed `dynamic_bitset` that is a subset of the `dynamic_bitset` *x*, where the first *M* bits are initialized to bits `[`*pos*`,`*pos*`+`*M*`)` in *x*, where *M* `=` `min(`*n*`, `*x*`.size()-`*pos*`, `*num_bits*`)` and the remaining bits, up to *num_bits*`-1` are initialized to `0`. If *n* is `npos` then *n* is considered to be *x*`.size()-`*pos*. If *num_bits* is `npos` then *num_bits* is considered to be `min(`*n*`, `*x*`.size()-`*pos*`)`.

Postconditions:

- `this->size()` == *num_bits*.
- For all *i* in the range `[0,`*M*`)`, `(*this)[`*i*`]` == *x*`[`*i*+*pos*`]`.
- For all *i* in the range `[`*M*`,`*num_bits*`)`, `(*this)[`*i*`]` == `false`.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

7 `dynamic_bitset` bitwise operations

`dynamic_bitset& operator&=(const dynamic_bitset&` *rhs*`);`

Effects: Bitwise-AND *M* bits in *rhs* with the corresponding bits in this `dynamic_bitset`, where *M* = `min(this->size(),` *rhs*`.size())`. If `this->size()` is greater than *rhs*`.size()`, then the remaining `this->size()` - *M* bits are bitwise-ANDed with `0`. This computes the set intersection of this `dynamic_bitset` and the *rhs* `dynamic_bitset`. The effect is to clear each bit in `*this` for which the corresponding bit in the *rhs* is clear, leaving all other bits unchanged. This is equivalent to:

```
for (size_type i = 0; i != M; ++i)
  (*this)[i] = (*this)[i] & rhs[i];
for (size_type i = M; i != this->size(); ++i)
  (*this)[i] = (*this)[i] & 0;
```

Returns: `*this`.

Throws: nothing.

`dynamic_bitset& operator|=(const dynamic_bitset&` *rhs*`);`

Effects: Bitwise-OR *M* bits in *rhs* with the corresponding bits in this `dynamic_bitset`, where *M* = `min(this->size(),` *rhs*`.size())`. If `this->size()` is greater than *rhs*`.size()`, then the remaining `this->size()` - *M* bits are bitwise-ORed with `0`. This computes the set intersection of this `dynamic_bitset` and the *rhs* `dynamic_bitset`. The effect is to clear each bit in `*this` for which the corresponding bit in the *rhs* is clear, leaving all other bits unchanged. This is equivalent to:

```
for (size_type i = 0; i != M; ++i)
  (*this)[i] = (*this)[i] | rhs[i];
for (size_type i = M; i != this->size(); ++i)
  (*this)[i] = (*this)[i] | 0;
```

Returns: `*this`.

Throws: nothing.

`dynamic_bitset& operator^=(const dynamic_bitset&` *x*`);`

Effects: Bitwise-XOR's M bits in *rhs* with the bits in this `dynamic_bitset`, where *M* = `min(this->size(),` *rhs*`.size())`. If `this->size()` is greater than *rhs*`.size()`, then the remaining `this->size()` - *M* bits are bitwise-XORed with `0`. The effect is to toggle each bit in `*this` for which the corresponding bit in the *rhs* is set, leaving all other bits unchanged. This is equivalent to:

```
for (size_type i = 0; i != M; ++i)
  (*this)[i] = (*this)[i] ^ rhs[i];
```

```
for (size_type i = M; i != this->size(); ++i)
  (*this)[i] = (*this)[i] ^ 0;
```

Returns: `*this`.

Throws: nothing.

`dynamic_bitset& operator-=(const dynamic_bitset& x);`

Effects: Bitwise-AND the complement of *M* bits in *rhs* with the corresponding bits in this `dynamic_bitset`, where *M* `= min(this->size(), rhs.size())`. If `this->size()` is greater than *rhs*`.size()`, then the remaining `this->size() -` *M* bits are bitwise-ANDed with `1`. This computes the set difference of this `dynamic_bitset` and the *rhs* `dynamic_bitset`. This is equivalent to:

```
for (size_type i = 0; i != M; ++i)
  (*this)[i] = (*this)[i] & ~rhs[i];
for (size_type i = M; i != this->size(); ++i)
  (*this)[i] = (*this)[i] & 1;
```

Returns: `*this`.

Throws: nothing.

8 `dynamic_bitset` bit shift operations

`dynamic_bitset& operator<<=(size_type n);`

Effects: Replaces each bit position *pos* in `*this` with a value determined as follows:

— If *pos* `<` *n*, the new value is zero;

— If *pos* `>=` *n*, the new value is the previous value of the bit at position *pos* `-` *n*.

Returns: `*this`.

Throws: nothing.

`dynamic_bitset& operator>>=(size_type n);`

Effects: Replaces each bit position *pos* in `*this` with a value determined as follows:

— If *n* `>= size() -` *pos*, the new value is zero;

— If *n* `< size() -` *pos*, the new value is the previous value of the bit at position *pos* `+` *n*.

Returns: `*this`.

Throws: nothing.

`dynamic_bitset operator<<(size_type n) const;`

Effects: Constructs an object *x* of `dynamic_bitset` and initialises it with `*this`.

Returns: *x* `<<=` *n*.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

`dynamic_bitset operator>>(size_type` *n*`) const;`

Effects: Constructs an object *x* of `dynamic_bitset` and initialises it with `*this`.

Returns: *x* `>>=` *n*.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

9 `dynamic_bitset` basic bit operations

`dynamic_bitset& replace(size_type` *pos*`, bool` *val*`);`

Requires: *pos* `< this->size()`.

Effects: Sets bit *pos* in `*this` to `1` if *val* is `true`, and clears bit *pos* in `*this` to `0` if *val* is `false`.

Returns: `*this`.

Throws: nothing.

`dynamic_bitset& set(size_type` *pos*`);`

Requires: *pos* `< this->size()`.

Effects: Sets bit *pos* in `*this` to `1`.

Returns: `*this`.

Throws: nothing.

`dynamic_bitset& set();`

Effects: Sets all bits in `*this` to `1`.

Returns: *this.

Throws: nothing.

`dynamic_bitset& reset(size_type` *pos*`);`

Requires: *pos* `< this->size()`.

Effects: Sets bit *pos* in `*this` to `0`.

Returns: `*this`.

Throws: nothing.

`dynamic_bitset& reset();`

Effects: Clears all bits in `*this` to `0`.

Returns: `*this`.

Throws: nothing.

`dynamic_bitset& flip(size_type` *pos*`);`

Requires: *pos* `< this->size()`.

Effects: Toggles the bit at *pos* in `*this`.

Returns: `*this`.

Throws: nothing.

```
dynamic_bitset& flip();
```

Effects: Toggles all bits in `*this`.

Returns: `*this`.

Throws: nothing.

```
dynamic_bitset operator~() const;
```

Effects: Constructs an object *x* of `dynamic_bitset` and initialises it with `*this`.

Returns: *x*`.flip()`.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

```
bool test(size_type pos) const;
```

Requires: *pos* `< this->size()`.

Returns: `true` if the bit at *pos* in `*this` has the value `1`, `false` otherwise.

Throws: nothing.

```
bool any() const;
```

Returns: true if any bit in **this is 1, false otherwise.

Throws: nothing.

```
bool none() const;
```

Returns: `true` if no bit in `*this` is `1`, `false` otherwise.

Throws: nothing.

```
bool all() const;
```

Returns: `true` if all bits in `*this` are `1`, `false` otherwise.

Throws: nothing.

```
size_type count() const;
```

Returns: A count of the number of bits set to `1` in `*this`.

Throws: nothing.

10 `dynamic_bitset` element access

```
reference operator[](size_type pos);
```

Requires: *pos* `< this->size()`.

Returns: A `reference` to bit *pos*. Note that `reference` is a proxy class with an assignment operator and a conversion to `bool`, which allows you to use `operator[]` for assignment. That is, if *x* is a variable convertible to `bool`, you can write both *x* `= b[`*pos*`]` and `b[`*pos*`] =` *x*. However, in many other respects the proxy is not the same as the true reference type `bool&`.

Throws: nothing.

```
bool operator[](size_type pos) const;
```

Requires: *pos* `< this->size()`.

Returns: `test(`*pos*`)`.

Throws: nothing.

11 `dynamic_bitset` conversion

```
unsigned long to_ulonglong() const;
```

Returns: The numeric value corresponding to the bits in `*this`.

Throws: `std::overflow_error` if that value is too large to be represented in an `unsigned long long`, i.e. if `*this` has any non-zero bit at a position `>= std::numeric_limits<unsigned long long>::digits`.

12 `dynamic_bitset` capacity

```
size_type size() const;
```

Returns: the number of bits in this `dynamic_bitset`.

Throws: nothing.

```
size_type num_blocks() const;
```

Returns: the number of blocks in this `dynamic_bitset`.

Throws: nothing.

```
size_type max_size() const;
```

Returns: `size()` of the largest possible `dynamic_bitset`.

Throws: nothing.

```
bool empty() const;
```

Returns: `size() == 0`.

Throws: nothing.

13 `dynamic_bitset` set queries

```
bool is_subset_of(const dynamic_bitset& x,
```

```
                int offset = 0,
                size_type n = npos) const;
```

Returns: `true` if this `dynamic_bitset` is a subset of *n* bits of `dynamic_bitset` *x* offset by *offset*. That is, it returns `true` if, for every bit *pos* that is set to `1` in this `dynamic_bitset`, the corresponding bit *pos*+*offset* in `dynamic_bitset` *x* is also set to `1`. Otherwise this function returns `false`. Positions in `dynamic_bitset` *x* less than `0` or greater than *pos*+*offset*+*n* are considered to be `0`.

[*Example:* If `*this == dynamic_bitset(string("11101101"),false)` and *x* == `dynamic_bitset(string("101111011010"),false)` then, `this->is_subset_of(`*x*`, -2, 4) == true` but `this->is_subset_of(`*x*`, 3) == false`.
—*end example*]

Throws: nothing.

```
bool is_proper_subset_of(const dynamic_bitset& x,
                         int offset = 0,
                         size_type n = npos) const;
```

Returns: `true` if this `dynamic_bitset` is a proper subset of *n* bits of `dynamic_bitset` *x* offset by *offset*. That is, it returns `true` if, for every bit that is set to `1` in this `dynamic_bitset`, the corresponding bit in `dynamic_bitset` *x* is also set to `1` and `this->count() < x.count`. Otherwise this function returns `false`.

[*Example:* If `*this == dynamic_bitset(string("010010101101")), this->count() == 6`; and *x* `== dynamic_bitset(string("011011101101")), x.count() == 8`; then, `this->is_proper_subset_of(x) == true`.
—*end example*]

Throws: nothing.

```
bool is_superset_of(const dynamic_bitset& x,
                    int offset = 0,
                    size_type n = npos) const;
```

Returns: `x.is_subset_of(*this,`*offset* `,`*n* `);`

Throws: nothing.

```
bool is_proper_superset_of(const dynamic_bitset& x,
                           int offset = 0,
                           size_type n = npos) const;
```

Returns: `x.is_proper_subset_of(*this,`*offset* `,`*n* `);`

Throws: nothing.

14 `dynamic_bitset` lookup

```
size_type find_first_set() const;
```

Returns: the lowest index *i* such as bit *i* is set to `1`, or `npos` if `*this` has no one bits.

Throws: nothing.

`size_type find_next_set(size_type` *pos*`) const;`

Requires: *pos* `< this->size()`.

Returns: the lowest index *i* greater than *pos* such as bit *i* is set to `1`, or `npos` if no such index exists.

Throws: nothing.

`size_type find_last_set() const;`

Returns: the highest index *i* such as bit *i* is set to `1`, or `npos` if `*this` has no one bits.

Throws: nothing.

`size_type find_prev_set(size_type` *pos*`) const;`

Requires: *pos* `< this->size()`.

Returns: the highest index *i* less than *pos* such as bit *i* is set to `1`, or `npos` if no such index exists.

Throws: nothing.

`size_type find_first_reset() const;`

Returns: the lowest index *i* such that bit *i* is clear (`0`), or `npos` if `*this` has no zero bits.

Throws: nothing.

`size_type find_next_reset(size_type` *pos*`) const;`

Requires: *pos* `< this->size()`.

Returns: the lowest index *i* greater than *pos* such that bit *i* is clear (`0`), or `npos` if no such index exists.

Throws: nothing.

`size_type find_last_reset() const;`

Returns: the highest index *i* such that bit *i* is clear (`0`), or `npos` if `*this` has no zero bits.

Throws: nothing.

`size_type find_prev_reset(size_type` *pos*`) const;`

Requires: *pos* `< this->size()`.

Returns: the highest index *i* less than *pos* such as bit *i* is clear (`0`), or `npos` if no such index exists.

Throws: nothing.

15 `dynamic_bitset` non-member operations

```
template <typename Block,
          typename Allocator>
void swap(dynamic_bitset<Block, Allocator>& lhs,
          dynamic_bitset<Block, Allocator>& rhs);
```

Returns: *lhs*`.swap(`*rhs*`);`

Throws: nothing.

```
template <typename Block,
          typename Allocator>
bool operator==(const dynamic_bitset<Block, Allocator>& lhs,
                const dynamic_bitset<Block, Allocator>& rhs);
```

Returns: `true` if *lhs*`.size() ==` *rhs*`.size()` and if for all *i* in the range `[0,`*rhs*`.size()),` `(*this)[i] ==` *rhs*`[i].` Otherwise returns `false`.

Throws: nothing.

```
template <typename Block,
          typename Allocator>
bool operator!=(const dynamic_bitset<Block, Allocator>& lhs,
                const dynamic_bitset<Block, Allocator>& rhs);
```

Returns: `!(`*lhs* `==` *rhs*`);`

Throws: nothing.

```
template <typename Block, typename Allocator>
bool operator<(const dynamic_bitset<Block, Allocator>& lhs,
               const dynamic_bitset<Block, Allocator>& rhs);
```

Returns: `true` if *lhs* is lexicographically less than *rhs*, `false` otherwise.

Throws: nothing.

```
template <typename Block,
          typename Allocator>
bool operator<=(const dynamic_bitset<Block, Allocator>& lhs,
                const dynamic_bitset<Block, Allocator>& rhs);
```

Returns: `!(`*lhs* `>` *rhs*`);`

Throws: nothing.

```
template <typename Block,
          typename Allocator>
bool operator>(const dynamic_bitset<Block, Allocator>& lhs,
               const dynamic_bitset<Block, Allocator>& rhs);
```

Returns: `!(`*lhs* `<` *rhs* `||` *lhs* `==` *rhs*`);`

Throws: nothing.

```
template <typename Block,
          typename Allocator>
bool operator>=(const dynamic_bitset<Block, Allocator>& lhs,
                const dynamic_bitset<Block, Allocator>& rhs);
```

Returns: `!(`*lhs* `<` *rhs*`);`

Throws: nothing.

```
template <typename Block,
          typename Allocator>
dynamic_bitset<Block, Allocator>
operator&(const dynamic_bitset<Block, Allocator>& lhs,
          const dynamic_bitset<Block, Allocator>& rhs);
```

Returns: `dynamic_bitset<Block,Allocator>(`*lhs*`) &= `*rhs*.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

```
template <typename Block,
          typename Allocator>
dynamic_bitset<Block, Allocator>
operator|(const dynamic_bitset<Block, Allocator>& lhs,
          const dynamic_bitset<Block, Allocator>& rhs);
```

Returns: `dynamic_bitset<Block,Allocator>(`*lhs*`) |= `*rhs*.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

```
template <typename Block,
          typename Allocator>
dynamic_bitset<Block, Allocator>
operator^(const dynamic_bitset<Block, Allocator>& lhs,
          const dynamic_bitset<Block, Allocator>& rhs);
```

Returns: `dynamic_bitset<Block,Allocator>(`*lhs*`) ^= `*rhs*.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

```
template <typename Block,
          typename Allocator>
dynamic_bitset<Block, Allocator>
operator-(const dynamic_bitset<Block, Allocator>& lhs,
          const dynamic_bitset<Block, Allocator>& rhs);
```

Returns: `dynamic_bitset<Block,Allocator>(`*lhs*`) -= `*rhs*.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`).

```
template <typename Block,
          typename Allocator>
bool intersect(const dynamic_bitset<Block, Allocator>& lhs,
               const dynamic_bitset<Block, Allocator>& rhs,
               int offset = 0);
```

Effect: `true` if there exists any valid position *pos* in *lhs* and any valid position *pos*+*offset* in *rhs* for

which *lhs*[*pos*] & *rhs*[*pos*+*offset*] is true. No new dynamic_bitset is created.

Returns: dynamic_bitset<Block, Allocator>(*lhs* & dynamic_bitset<Block, Allocator>(*rhs*, *offset*)).any();

Throws: nothing.

```
template <typename Block,
          typename Allocator>
bool disjoint(const dynamic_bitset<Block, Allocator>& lhs,
              const dynamic_bitset<Block, Allocator>& rhs,
              int offset = 0);
```

Effect: true if there exists no valid position *pos* in both *lhs* and valid position *pos*+*offset* in *rhs* for which *lhs*[*pos*] & *rhs*[*pos*+*offset*] is true.

Returns: !intersect(lhs, rhs, offset);

Throws: nothing.

```
template <typename Block,
          typename Allocator,
          typename StringT>
void to_string(const dynamic_bitset<Block, Allocator>& x,
               StringT& str,
               bit_order order = msb_first);
```

Effects: Copies a representation of *x* into the string *str*. A character in the string is '1' if the corresponding bit is set to 1, and '0' if it is not. If *order* is msb_first the character position *pos* in the string corresponds to bit position *x*.size() - 1 - *pos*, otherwise it corresponds to *pos*.

Throws: An allocation error if memory is exhausted in *str*.

```
template <typename Block,
          typename Allocator,
          typename BlockOutputIterator>
void to_block_range(const dynamic_bitset<Block, Allocator>& x,
                    BlockOutputIterator result);
```

Requires: The type BlockOutputIterator must be a model of Output Iterator and its value_type must be the same type as Block. Further, the size of the output range must be greater or equal *x*.num_blocks().

Effects: Writes the bits of the dynamic_bitset into the iterator result a block at a time. The first block written represents the bits in the position range [0,bits_per_block) in the dynamic_bitset, the second block written the bits in the range [bits_per_block,2*bits_per_block), and so on. For each block *blockvalue* written, the bit (*blockvalue* >> *i*) & 1 corresponds to the bit at position (*blockvalue* * bits_per_block + *i*) in the dynamic_bitset.

```
template <typename CharT,
          typename Traits,
          typename Block,
          typename Allocator>
```

```
std::basic_ostream<CharT, Traits>&
operator<<(std::basic_ostream<CharT, Traits>& os,
           const dynamic_bitset<Block, Allocator>& x);
```

Effects: Inserts a textual representation of *x* into the stream *os* (highest bit first). Informally, the output is the same as doing,

```
std::basic_string<CharT, Traits> str;
to_string(x, str);
os << str;
```

except that the stream inserter takes into account the locale imbued into *os*, which `to_string()` cannot. More precisely, assume we have `character_of(`*x*`[`*pos*`])` for each valid *pos* in *x* where `character_of(`*bit*`) = `*bit*` ? `*os*`.widen('1') : `*os*`.widen('0')`. Then assume we have *str* of type `std::basic_string<CharT, Traits>` and length *x*`.size()`, such that for each *pos* in `[0,` *x*`.size())`, *str*`[`*pos*`]` is `character_of(`*x*`[`*pos*`])`. Then, the output, the effects on *os* and the exception behaviour, is the same as outputting the object *str* to *os* (same width, same exception mask, same padding, same `setstate()` logic).

Returns: os

Throws: `std::ios_base::failure` if there is a problem writing to the stream.

```
template <typename CharT,
          typename Traits,
          typename Block,
          typename Allocator>
std::basic_istream<CharT, Traits>&
operator>>(std::basic_istream<CharT, Traits>& is,
           dynamic_bitset<Block, Allocator>& x);
```

Effects: Extracts a `dynamic_bitset` from an input stream. Let *tt* be the `traits_type` of *is*. Then:

1. A (non-eof) character *c* extracted from *is* is a *bitset_digit* if, and only if, either *tt*`::eq(`*c*`,` *is*`.widen('0'))` or *tt*`::eq(`*c*`,` *is*`.widen('1'))` return `true`.
2. If *c* is a *bitset_digit*, its *corresponding_bit_value* is `0` if *tt*`::eq(`*c*`,` *is*`.widen('0'))` is `true`, `1` otherwise.

The function begins by constructing a sentry object *k* as if *k* were constructed by `typename std::basic_istream<CharT, Traits>::sentry k(`*is*`)`. If `bool(`*k*`)` is `true`, it calls *x*`.clear()` then attempts to extract characters from *is*. For each character *c* that is a *bitset_digit* the *corresponding_bit_value* is appended to the less significant end of *x*. If *is*`.width()` is greater than zero and smaller than *x*`.max_size()` then the maximum number, *n*, of bits appended is *is*`.width()` otherwise *n* = *x*`.max_size()`. Unless the extractor is exited via an exception, characters are extracted (and *corresponding_bit_value*s appended) until any of the following occurs:

— *n* bits are stored into the `dynamic_bitset`;

— end-of-file, or an error, occurs on the input sequence;

— the next available input character isn't a *bitset_digit*

If no exception caused the function to exit then *is*`.width(0)` is called, regardless of how many characters were actually extracted. The sentry object *k* is destroyed.

If the function extracts no characters, it calls *is*`.setstate(std::ios::failbit)`, which may throw `std::ios_base::failure`.

Returns: *is*

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if `Allocator` is `std::allocator`). A `std::ios_base::failure` if there is a problem reading from the stream.

16 `set_bit_order` constructor

```
explicit
set_bit_order(bit_order o);
```

Effects: Constructs a `set_bit_order` object with `bit_order` *o*.

Throws: Nothing.

17 `dynamic_bitset` bit order stream manipulators

```
template<class CharT, class Traits>
std::basic_ostream<CharT, Traits>& operator<<
  (std::basic_ostream<CharT, Traits>& os, const set_bit_order& sbo);
```

Effects: Ensures that subsequent output streaming of `dynamic_bitset`s will be bit ordered according the value of the `bit_order` used to construct *sbo*.

Returns: *os*

Throws: Nothing.

V Future Issues and Discussion

1 Iterators

`dynamic_bitset` is not designed to be a container and does not provide iterators. Primarily this is because the current iterator requirements for a Random Access Iterator does not allow the use of proxies. However if the possible changes outlined in [N1640 – Abrahams, Siek, Witt, 2004] are adopted, then it is conceivable that iterators might be provided for `dynamic_bitset` in a fashion similar to that posited for `std::vector<bool>`, namely iterators that meet the requirements of Random Access Traversal Iterator, Readable Iterator and Writeable Iterator would be feasible.

VI Acknowledgments

Many thanks go to the original creators and maintainers of the `boost::dynamic_bitset` library on which this proposal is based, namely, Jeremy Siek and Chuck Allison..

VII References

[Boost – dynamic_bitset] Boost Library documentation for the boost::dynamic_bitset library.
http://www.boost.org/libs/dynamic_bitset/dynamic_bitset.html

[Boost Libraries] Boost provides free peer-reviewed portable C++ source libraries.
http://www.boost.org/

[Gmane – boost.devel] The Boost Developers mailing list at gmane.comp.lib.boost.devel.
nntp://news.gmane.org/gmane.comp.lib.boost.devel
http://news.gmane.org/gmane.comp.lib.boost.devel

[Meyers, 2001] "Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library", Scott Meyers, 2001, Addison-Wesley. pp79 – 82.

[N0220 – Allison, 1993] "A Proposal for Two Bitset Classes", Chuck Allison, 8th Dec. 1993. Library.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1993/N0220R2.asc

[N1185 – Sutter, 1999] N1185: 99-0008, "vector<bool> Is Nonconforming, and Forces Optimization Choice", Herb Sutter, 22nd Feb. 1999. Library.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1999/n1185.pdf

[N1640 – Abrahams, Siek, Witt, 2004] N1640: 04-0080, "New Iterator Concepts", David Abrahams, Jeremy Siek, Thomas Witt, 10th Apr. 2004. Library.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1640.html

[N1847 – Sutter, 1999] N1847: 05-0107, "vector<bool>: More Problems, Better Solutions", Herb Sutter, 20th Oct. 1999. Library.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1847.pdf

[N2009 – Becker, 2006] N2009: 06-0079, "Working Draft, Standard for Programming Language C++", Pete Becker, 21st Apr. 2006.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2009.pdf

[Sutter, 2005] "Exception C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions", Herb Sutter, 2005, Addison-Wesley. pp.204 – 211.