Wording for fundamental bit manipulation utilities

Note: this is an early draft. It’s known to be incomplet and incorrekt, and it has lots of bad formatting.
1 Bit manipulation library

1.1 General

This Clause describes the contents of the header `<bit>` (1.2) that provides components that C++ programs may use to access, manipulate, and process both individual bits and bit sequences.

The bit library relies on four main classes `bit_value` (1.4), `bit_reference` (1.5), `bit_pointer` (1.6) and `bit_iterator` (1.7) as well as on a helper class `binary_digits` (1.3). For generic purposes, `bit_value` and `bit_reference` exhibit roughly the same interface. Most of the non-member operations on `bit_value` (1.4.9) are provided on `bit_reference` through an implicit conversion to `bit_value` also accept `bit_reference`.

3 In all the following in this clause, a `bit` refers to an object, a hypothetical object that can hold one of the two values designated as 0 and 1. As a part of the C++ memory model, `CHAR_BIT` bits are packed together in `bytes`, with `CHAR_BIT` >= 8. Bytes are themselves packed together to form `machine words`. Because the smallest addressable entities in memory are bytes in the C++ memory model, a bit object is hypothetical.

4 An object of a `word` type refers to an object that provides access to its underlying bits. An object of a `word` type shall provide the operators `>>` and `&` such that the expression `(word >> position) & static_cast<decltype(word)>(1)` is a valid expression, with `word` an object of a `word` type and `position` a value of type `size_t` which is a valid bit index.

5 `binary_digits_v<WordType>` corresponds to the number of individual bits within a `word` of type `WordType`. The bit library is only compatible with `word` types `WordType` for which `binary_digits_v<WordType>` is defined and is not zero (1.3). `binary_digits_v<WordType>` corresponds to the number of individual bits within a `word` of type `WordType`.

6 The `position` of a bit within a `word` is the unsigned integral number in \([0, binary_digits_v<decltype(word)>]\) such that `(word >> n) & static_cast<decltype(word)>(1)` returns the `n`-th bit of the `word` [Note: For unsigned integral types, `(word >> n) & static_cast<decltype(word)>(1)` is equivalent to `word & (static_cast<decltype(word)>(1) << n)` for `n < binary_digits_v<decltype(word)>` — end note]

7 The least significant bit of a `word`, or lsb, is the bit at position 0. The most significant bit of a `word`, or msb, is the bit at position `binary_digits_v<WordType> - 1`.

8 The default direction in which bits are iterated through goes from the least significant bit to the most significant bit of each `word`. For purposes of iteration the next bit after the most significant bit of a `word` is the least significant bit of the next `word`. [Note: The arithmetic of bit pointers (1.6.1) and bit iterators (1.7.1) is based on this relationship. — end note]

1.2 Header `<bit>` synopsis

namespace std {

    // 1.3, helper class binary_digits
    template <class T> struct binary_digits;
    template <class T> inline constexpr std::size_t binary_digits_v = binary_digits<T>::value;

    // 1.4, class bit_value
    class bit_value;

    // 1.4.9, bit_value operations
    constexpr bit_value operator=(bit_value rhs) noexcept;

} § 1.2
constexpr bit_value operator&(bit_value lhs, bit_value rhs) noexcept;
constexpr bit_value operator|(bit_value lhs, bit_value rhs) noexcept;
constexpr bit_value operator^(bit_value lhs, bit_value rhs) noexcept;

// 1.4.9, bit_value comparisons
constexpr bool operator==(bit_value lhs, bit_value rhs) noexcept;
constexpr bool operator!=(bit_value lhs, bit_value rhs) noexcept;
constexpr bool operator<(bit_value lhs, bit_value rhs) noexcept;
constexpr bool operator<=(bit_value lhs, bit_value rhs) noexcept;
constexpr bool operator>(bit_value lhs, bit_value rhs) noexcept;
constexpr bool operator>=(bit_value lhs, bit_value rhs) noexcept;

// 1.4.9, bit_value input and output
template <class charT, class traits>
basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>& is, bit_value& x);

template <class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& os, bit_value x);

// 1.5, class template bit_reference
template <class WordType> class bit_reference;

// 1.5.9, bit_reference swap
template <class T>
void swap(bit_reference<T> lhs, bit_reference<T> rhs) noexcept;

template <class T, class U>
void swap(bit_reference<T> lhs, bit_reference<U> rhs) noexcept;

template <class T>
void swap(bit_reference<T> lhs, bit_value& rhs) noexcept;

template <class U>
void swap(bit_value& lhs, bit_reference<U> rhs) noexcept;

// 1.5.9, bit_reference input and output
template <class charT, class traits, class T>
basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>& is, bit_reference<T>& x);

template <class charT, class traits, class T>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& os, bit_reference<T> x);

// 1.6, class template bit_pointer
template <class WordType> class bit_pointer;

// 1.6.7, bit_pointer arithmetic
template <class T>
constexpr bit_pointer<T> operator+(typename bit_pointer<T>::difference_type n, bit_pointer<T> x);

template <class T, class U>
constexpr common_type_t<typename bit_pointer<T>::difference_type, typename bit_pointer<U>::difference_type>
template<typename bit_pointer<U>::difference_type
> operator-(bit_pointer<T> lhs, bit_pointer<U> rhs);

// 1.6.7, bit_pointer comparison
template <class T, class U>
constexpr bool operator==(bit_pointer<T> lhs, bit_pointer<U> rhs) noexcept;
template <class T, class U>
constexpr bool operator!=(bit_pointer<T> lhs, bit_pointer<U> rhs) noexcept;
template <class T, class U>
constexpr bool operator<(bit_pointer<T> lhs, bit_pointer<U> rhs) noexcept; \added{not no except because narrow}
template <class T, class U>
constexpr bool operator<=(bit_pointer<T> lhs, bit_pointer<U> rhs) noexcept; \added{not no except because narrow}
template <class T, class U>
constexpr bool operator>(bit_pointer<T> lhs, bit_pointer<U> rhs) noexcept; \added{not no except because narrow}
template <class T, class U>
constexpr bool operator>=(bit_pointer<T> lhs, bit_pointer<U> rhs) noexcept; \added{not no except because narrow}

// 1.7, class template: bit_iterator
template <class Iterator> class bit_iterator;

// 1.7.7, bit_iterator arithmetic
template <class T>
constexpr bit_iterator<T> operator+(typename bit_iterator<T>::difference_type n, const bit_iterator<T>& i);

template <class T, class U>
constexpr common_type_t<typename bit_iterator<T>::difference_type,
typename bit_iterator<U>::difference_type>
operator-(const bit_iterator<T>& lhs, const bit_iterator<U>& rhs);

// 1.7.7, bit_iterator comparisons
template <class T, class U>
constexpr bool operator==(const bit_iterator<T>& lhs, const bit_iterator<U>& rhs);
template <class T, class U>
constexpr bool operator!=(const bit_iterator<T>& lhs, const bit_iterator<U>& rhs);
template <class T, class U>
constexpr bool operator<(const bit_iterator<T>& lhs, const bit_iterator<U>& rhs);
template <class T, class U>
constexpr bool operator<=(const bit_iterator<T>& lhs, const bit_iterator<U>& rhs);
template <class T, class U>
constexpr bool operator>(const bit_iterator<T>& lhs, const bit_iterator<U>& rhs);
template <class T, class U>
constexpr bool operator>=(const bit_iterator<T>& lhs, const bit_iterator<U>& rhs);
}

1.3 Helper class binary_digits

1.3.1 Class binary_digits overview

template <class UIntType> struct binary_digits
: integral_constant<size_t, numeric_limits<UIntType>::digits> { };

1 Requires: UIntType is an unsigned integer type, otherwise the program is ill-formed. [Note: This
excludes bool and char. — end note]

2 Remarks: Specialization of this helper class for a type T informs other library components that this
type T corresponds to a word type whose bits can be accessed through bit_value, bit_reference,
bit_pointer, and bit_iterator.

### 1.3.2 Class binary_digits specializations  

```
template <class T> struct binary_digits<const T>; \added{inherit from something see tuple}
template <class T> struct binary_digits<volatile T>; \added{inherit from something see tuple}
template <class T> struct binary_digits<const volatile T>; \added{inherit from something see tuple}
```

The value of each member of a specialization of `binary_digits` on a cv-qualified type `cv T` shall be equal to the value of the corresponding member of the specialization on the unqualified type `T`.

```
template <> struct binary_digits<byte>
: integral_constant<size_t, numeric_limits<unsigned char>::digits> { };
```

### 1.4 Class bit_value

#### 1.4.1 Class bit_value overview

A `bit_value` emulates the behavior of an independent single bit, with no arithmetic behavior apart from bitwise compound assignment (1.4.5) and bitwise operators (1.4.9). It provides the bit modifier members `set`, `reset`, and `flip` (1.4.7). [Note: A `bit_value` can be implemented as a wrapper around `bool`. — end note]

A `bit_value` is implicitly convertible from a `bit_reference` (1.5).

To prevent implicit conversions to `bool` and `int` potentially leading to misleading arithmetic behaviors, a `bit_value` is explicitly convertible to `bool` (1.4.6).

```
class bit_value {
public:
  // 1.4.2, types
  using size_type = see below;

  // 1.4.3, constructors
  bit_value() noexcept = default;
  template <class T> constexpr bit_value(bit_reference<T> ref) noexcept;
  template <class WordType> explicit constexpr bit_value(WordType val) noexcept;
  template <class WordType> constexpr bit_value(WordType val, size_type pos);

  // 1.4.4, assignment
  template <class T> constexpr bit_value& operator=(bit_reference<T> r) noexcept;
  template <class WordType> constexpr bit_value& assign(WordType val) noexcept;
  template <class WordType> constexpr bit_value& assign(WordType val, size_type pos);

  // 1.4.5, compound assignment
  constexpr bit_value& operator&=(bit_value rhs) noexcept;
  constexpr bit_value& operator|=(bit_value rhs) noexcept;
  constexpr bit_value& operator^=(bit_value rhs) noexcept;

  // 1.4.6, observers
  explicit constexpr operator bool() const noexcept;

  // 1.4.7, modifiers
  constexpr bit_value& set(bool b) noexcept;
  constexpr bit_value& set() noexcept;
  constexpr bit_value& reset() noexcept;
  constexpr bit_value& flip() noexcept;

  // 1.4.8, swap
  void swap(bit_value& rhs) noexcept;
};
```
template <class T> void swap(bit_reference<T> rhs) noexcept;
;
1.4.2 bit_value member types

using size_type = see below;

Type: An implementation-defined unsigned integer type capable of holding at least as many values as binary_digits_v<std::uintmax_t>. Same as std::remove_cv<decltype(binary_digits_v<std::uintmax_t>)>::type (1.3).

1.4.3 bit_value constructors

bit_value() noexcept = default;

Effects: Constructs a default-initialized object of type bit_value. (circular definition: see default duration. 'The resulting object has a valid but unspecified value.')

template <class T> constexpr bit_value(bit_reference<T> ref) noexcept;

Effects: Constructs an object of type bit_value from the value of the bit referenced by ref.

template <class WordType> explicit constexpr bit_value(WordType val) noexcept;

Effects: Constructs an object of type bit_value from the value of the bit in val at position 0.

Remarks: This constructor shall not participate in overload resolution unless binary_digits_v<WordType> is well-defined and is not equal to zero (1.3).

template <class WordType> constexpr bit_value(WordType val, size_type pos);

Requires: pos < binary_digits_v<WordType>.

Effects: Assigns the value of the bit in val at position pos to *this.

Returns: *this.

Remarks: This constructor shall not participate in overload resolution unless binary_digits_v<WordType> is well-defined and is not equal to zero (1.3).

1.4.4 bit_value assignment

template <class T> constexpr bit_value& operator=(bit_reference<T> ref) noexcept;

Effects: Assigns the value of the bit referenced by ref to *this.

Returns: *this.

template <class WordType> constexpr bit_value& assign(WordType val) noexcept;

Effects: Assigns the value of the bit in val at position 0 to *this.

Returns: *this.

(Requires: This function shall not participate in overload resolution unless binary_digits_v<WordType> is well-defined and is not equal to zero (1.3). and ill-formed unless)

template <class WordType> constexpr bit_value& assign(WordType val, size_type pos);

Requires: pos < binary_digits_v<WordType>.

Effects: Assigns the value of the bit in val at position pos to *this.

Returns: *this.

Throws: Nothing.
(Requires: This function shall not participate in overload resolution unless binary_digits_v<WordType>
is well-defined and is not equal to zero (1.3), and ill-formed unless)

1.4.5 bit_value compound assignment

constexpr bit_value& operator&=(bit_value rhs) noexcept;

   Effects: Sets the bit to zero if rhs is zero.
   Returns: *this.

constexpr bit_value& operator|=(bit_value rhs) noexcept;

   Effects: Sets the bit to one if rhs is one.
   Returns: *this.

constexpr bit_value& operator^=(bit_value rhs) noexcept;

   Effects: Toggles the bit if rhs is one.
   Returns: *this.

1.4.6 bit_value observers

explicit constexpr operator bool() const noexcept;

   Returns: false if the bit is zero, true if the bit is one.

1.4.7 bit_value modifiers

constexpr bit_value& set(bool b) noexcept;

   Effects: Stores a new value in the bit: one if b is true, zero otherwise.
   Returns: *this.

constexpr bit_value& set() noexcept;

   Effects: Sets the bit to one.
   Returns: *this.

constexpr bit_value& reset() noexcept;

   Effects: Resets the bit to zero.
   Returns: *this.

constexpr bit_value& flip() noexcept;

   Effects: Toggles the bit.
   Returns: *this.

1.4.8 bit_value swap

void swap(bit_value& rhs) noexcept;

   Effects: Toggles the bit stored in *this and the bit stored in rhs if their values differ as in static_cast<bool>(*this) != static_cast<bool>(rhs).

template <class T> void swap(bit_reference<T> rhs) noexcept;

   Effects: Toggles the bit stored in *this and the bit referenced by rhs if their value differ as in static_cast<bool>(*this) != static_cast<bool>(rhs).
1.4.9  bit_value non-member operations

constexpr bit_value operator~(bit_value x) noexcept;

Returns: bit_value(x).flip().

constexpr bit_value operator&(bit_value lhs, bit_value rhs) noexcept;

Returns: lhs & rhs.

constexpr bit_value operator|(bit_value lhs, bit_value rhs) noexcept;

Returns: lhs | rhs.

constexpr bit_value operator^(bit_value lhs, bit_value rhs) noexcept;

Returns: lhs ^= rhs.

constexpr bool operator==(bit_value lhs, bit_value rhs) noexcept;

Returns: static_cast<bool>(lhs) == static_cast<bool>(rhs).

constexpr bool operator!=(bit_value lhs, bit_value rhs) noexcept;

Returns: static_cast<bool>(lhs) != static_cast<bool>(rhs).

constexpr bool operator<(bit_value lhs, bit_value rhs) noexcept;

Returns: static_cast<bool>(lhs) < static_cast<bool>(rhs).

constexpr bool operator<=(bit_value lhs, bit_value rhs) noexcept;

Returns: static_cast<bool>(lhs) <= static_cast<bool>(rhs).

constexpr bool operator>(bit_value lhs, bit_value rhs) noexcept;

Returns: static_cast<bool>(lhs) > static_cast<bool>(rhs).

constexpr bool operator>=(bit_value lhs, bit_value rhs) noexcept;

Returns: static_cast<bool>(lhs) >= static_cast<bool>(rhs).

template <class charT, class traits>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& is, bit_value& x);

A formatted input function ([istream.formatted]).

Effects: A sentry object is first constructed. If the sentry object returns true, one character is extracted from is. If the character is successfully extracted with no end-of-file encountered, it is compared to is.widen('0') and to is.widen('1') and a temporary bit_value is set accordingly. If the character is neither equal to is.widen('0') nor to is.widen('1'), the extracted character is put back into the sequence. If the extraction succeeds, the temporary bit value is assigned to x, otherwise is.setstate(ios_base::failbit) is called (which may throw ios_base::failure).

Returns: is.

template <class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, bit_value x);

A formatted output function ([ostream.formatted.reqmts]).

Effects: Outputs the bit to the stream.

Returns: os << x ? '1' : '0'.
1.4.10 bit_value objects

inline constexpr bit_value bit0(0U);
inline constexpr bit_value bit1(1U);

Replace by the example

1 The object bit0 represents a constant bit of value 0.

1.5 Class template bit_reference

1.5.1 Class template bit_reference overview

1 A bit_reference emulates the behavior of a reference to a bit within an object, with no arithmetic behavior apart from bitwise compound assignment (1.5.5) and bitwise operators provided through implicit conversion to bit_value (1.4.9). Comparison operators are provided through implicit conversion to bit_value (1.4.9).

[Note: A bit_reference is typically implemented in terms of a bit position or a mask, and in terms of a pointer or a reference to the object in which the bit is referenced. — end note]

2 The copy assignment operator is overloaded to assign a new value to the referenced bit without changing the underlying reference itself. Specializations of swap are provided for the same reason, typically using a temporary bit_value (1.4) to ensure that the referenced values are swapped and not the references themselves.

3 The address-of operator of bit_reference (1.5.6) is overloaded to return a bit_pointer (1.6) to the referenced bit. [Note: A pointer to a bit_reference can be obtained through the addressof function of the standard library. — end note]

4 An access to the underlying representation of a bit_reference is provided through the function members address, position, and mask (1.5.6).

5 To prevent implicit conversions to bool and int potentially leading to misleading arithmetic behaviors, a bit_reference is explicitly, convertible to bool (1.5.6).

6 The template parameter type WordType shall be a type such that binary_digits_v<WordType> is well-defined and is not zero (1.3), otherwise the program is ill-formed. A reference to a constant bit can be obtained through bit_reference<const WordType>.

7 Concurrently reading and writing multiple bits belonging to the same underlying word through bit references may result in a data race.

For below, explore exposition only variables.

template <class WordType>
class bit_reference {
public:
  // 1.5.2, types
  using word_type = WordType;
  using size_type = see below; (replace with size_t)

  // 1.5.3, constructors
  // Add a copy constructor here = default
  template <class T> constexpr bit_reference(const bit_reference<T>& other) noexcept;
  explicit constexpr bit_reference(word_type& ref) noexcept;
  constexpr bit_reference(word_type& ref, size_type pos);

  // 1.5.4, assignment
  constexpr bit_reference& operator=(const bit_reference& other) noexcept;
  template <class T> constexpr bit_reference& operator=(const bit_reference<T>& other) noexcept;
  constexpr bit_reference& operator=(bit_value val) noexcept;
  constexpr bit_reference& assign(word_type val) noexcept;
constexpr bit_reference& assign(word_type val, size_type pos);

// 1.5.5, compound assignment
constexpr bit_reference& operator&=(bit_value rhs) noexcept;
constexpr bit_reference& operator|=(bit_value rhs) noexcept;
constexpr bit_reference& operator^=(bit_value rhs) noexcept;

// 1.5.6, observers
explicit constexpr operator bool() const noexcept;
constexpr bit_pointer<WordType> operator&() const noexcept;
constexpr word_type* address() const noexcept;
constexpr size_type position() const noexcept;
constexpr typename remove_cv_t<word_type> mask() const noexcept;

// 1.5.7, modifiers
constexpr bit_reference& set(bool b) noexcept;
constexpr bit_reference& set() noexcept;
constexpr bit_reference& reset() noexcept;
constexpr bit_reference& flip() noexcept;

// 1.5.8, swap
template <class T> void swap(bit_reference<T> rhs) noexcept;
void swap(bit_value& rhs) noexcept;

};

1.5.2 bit_reference member types                 [bit.reference.types]

using word_type = WordType;
1  Type: Refers to the underlying word type that is being provided as a template parameter.

using size_type = see below;
2  Type: An implementation-defined unsigned integer type capable of holding at least as many values as
binary_digits_v<word_type>. Same as bit_value::size_type (1.4.2).

1.5.3 bit_reference constructors                 [bit.reference.cons]

template <class T> constexpr bit_reference(const bit_reference<T>& other) noexcept;
1  Effects: Constructs an object of type bit_reference from another referenced bit other. [Note: This
constructor is typically used for implicit conversions of cv-qualified bit references. —end note]

Remarks: This constructor shall not participate in overload resolution unless is const_v<WordType>
is_same<T, remove_t<T> (check that: only work in the right case) the can be nis_convertible_v<T&,
word_type>&> == true.

explicit constexpr bit_reference(word_type& ref) noexcept;
3  Effects: Constructs a reference to the bit at position 0 of ref.

constexpr bit_reference(word_type& ref, size_type pos);
4  Requires: pos < binary_digits_v<word_type>.
5  Effects: Constructs a reference to the bit at position pos of ref.
6  Throws: Nothing.

1.5.4 bit_reference assignment                   [bit.reference.assign]

§ 1.5.4
constexpr bit_reference& operator=(const bit_reference& other) noexcept;
Effects: Assigns the value of the bit referenced by other to the bit referenced by *this.
Returns: *this.
[Note: The copy assignment operator is not implicitly generated in order to ensure that the value of the referenced bit is changed instead of the underlying reference itself. —end note]

template <class T> constexpr bit_reference& operator=(const bit_reference<T>& other) noexcept;
Effects: Assigns the value of the bit referenced by other to the bit referenced by *this.
Returns: *this.

castexpr bit_reference& operator=(bit_value val) noexcept;
Effects: Assigns the value of the bit val to the bit referenced by *this.
Returns: *this.

castexpr bit_reference& assign(word_type val) noexcept;
Effects: Assigns the value of the bit in val at position 0 to the bit referenced by *this.
Returns: *this.

castexpr bit_reference& assign(word_type val, size_type pos);
Requires: pos < binary_digits_v<word_type>.
Effects: Assigns the value of the bit in val at position pos to the bit referenced by *this.
Returns: *this.

1.5.5 bit_reference compound assignment
[bit.reference.cassign]
castexpr bit_reference& operator&=(bit_value rhs) noexcept;
Effects: Sets the bit referenced by *this to zero if rhs is zero.
Returns: *this.

castexpr bit_reference& operator|=(bit_value rhs) noexcept;
Effects: Sets the bit referenced by *this to one if rhs is one.
Returns: *this.

castexpr bit_reference& operator^=(bit_value rhs) noexcept;
Effects: Toggles the bit referenced by *this if rhs is one.
Returns: *this.

1.5.6 bit_reference observers
[bit.reference.observers]
explicit castexpr operator bool() const noexcept;
Returns: Returns the value of the bit reference. (make it the same as for bit_value. Need to be checked)

castexpr bit_pointer<WordType> operator&() const noexcept;
Returns: A bit_pointer (1.6) pointing to the bit referenced by *this.
[Note: The actual address of a bit_reference object can be obtained through the addressof function

§ 1.5.6
of the standard library. — end note]

constexpr word_type* address() const noexcept;

Returns: A pointer to the word containing the bit referenced by *this.

constexpr size_type position() const noexcept;

Returns: The position of the bit referenced by *this within the word containing it.

constexpr typename std::remove_cv<word_type>::type mask() const noexcept;

Returns: static_cast<word_type>(1) « position().

1.5.7 bit_reference modifiers

constexpr bit_reference& set(bool b) noexcept;

Effects: Stores a new value in the bit referenced by *this: one if b is true, zero otherwise.

Returns: *this.

constexpr bit_reference& set() noexcept;

Effects: Sets the bit referenced by *this to one.

Returns: *this.

constexpr bit_reference& reset() noexcept;

Effects: Resets the bit referenced by *this to zero.

Returns: *this.

constexpr bit_reference& flip() noexcept;

Effects: Toggles the bit referenced by *this.

Returns: *this.

1.5.8 bit_reference swap

template <class T> void swap(bit_reference<T> rhs) noexcept;

Effects: Toggles the bit referenced by *this and the bit referenced by rhs if their value differ as in static_cast<bool>(*this) != static_cast<bool>(rhs).

void swap(bit_value& rhs) noexcept;

Effects: Toggles the bit referenced by *this and the bit stored in rhs if their value differ as in static_cast<bool>(*this) != static_cast<bool>(rhs).

1.5.9 bit_reference non-member operations

template <class T> void swap(bit_reference<T> lhs, bit_reference<T> rhs) noexcept;

Effects: lhs.swap(rhs)

note Remarks: This overload of swap ensures that the values of the referenced bits are swapped instead of the underlying references themselves.

template <class T, class U> void swap(bit_reference<T> lhs, bit_reference<U> rhs) noexcept;

Effects: lhs.swap(rhs)

template <class T> void swap(bit_reference<T> lhs, bit_value& rhs) noexcept;
Effects: lhs.swap(rhs)

```cpp
template <class T> void swap(bit_value& lhs, bit_reference<T> rhs) noexcept;
```

Effects: lhs.swap(rhs)

```cpp
template <class charT, class traits, class T>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& is, bit_reference<T> x); (modify elsewhere)
```

A formatted input function ([istream.formatted]).

Effects: Assigns to bit value and set to bit reference

A sentry object is first constructed. If the sentry object returns true, one character is extracted from `is`. If the character is successfully extracted with no end-of-file encountered, it is compared to `is.widen('0')` and to `is.widen('1')` and a temporary `bit_value` is set accordingly. If the character is neither equal to `is.widen('0')` nor to `is.widen('1')`, the extracted character is put back into the sequence. If the extraction succeeds, the temporary bit value is assigned to `x`, otherwise `is.setstate(ios_base::failbit)` is called (which may throw `ios_base::failure`).

Returns: `is`.

```cpp
template <class charT, class traits, class T>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, bit_reference<T> x);
```

A formatted output function ([ostream.formatted.reqmts]).

Effects: Outputs the bit to the stream.

Returns: `os << os.widen(x ? '1' : '0')`.

1.6 Class template `bit_pointer` ([bit.pointer]

1.6.1 Class template `bit_pointer` overview ([bit.pointer.overview]

A `bit_pointer` emulates the behavior of a pointer to a bit within an object. [Note: A `bit_pointer` can be implemented in terms of a pointer to a `bit_reference` (1.5). — end note]

The indirection operator `*` of `bit_pointer` (1.6.5) is overloaded to return a `bit_reference` (1.5) to the pointed bit, while the arrow operator `->` is overloaded to return a pointer to a `bit_reference` (1.5). Bit modifiers (1.5.7) can be accessed through this interface, as well as the underlying representation (1.5.6).

A null bit pointer can be created from a null pointer (1.6.3). Dereferencing a null bit pointer leads to an undefined behavior. The explicit conversion to `bool` (1.6.5) shall return `false` for a null bit pointer, and `true` otherwise.

The arithmetic of bit pointers (1.6.6) rely on the ordering described in 1.1: a bit pointer `ptr2` is considered to be the next bit pointer of `ptr1` if both of them are not null and if either of the following is true:

\[
\begin{align*}
(4.1) & \quad \text{ptr2->address() - ptr1->address() == 0} \\
& \quad \text{&& ptr2->position() - ptr1->position() == 1}
\end{align*}
\]

\[
\begin{align*}
(4.2) & \quad \text{ptr2->address() - ptr1->address() == 1} \\
& \quad \text{&& binary_digits_v<typename decltype(ptr1)::word_type> - ptr1->position() == 1} \\
& \quad \text{&& ptr2->position() == 0}
\end{align*}
\]

Comparison operators for `bit_pointer` (1.6.7) rely on the same ordering, first comparing the addresses of the underlying values and then comparing bit positions in case of equality.

The template parameter type `WordType` should be a type such that `binary_digits_v<WordType>` is well-defined and is not zero (1.3), otherwise the program is ill-formed. A pointer to a constant bit shall be
obtained through \texttt{bit\_pointer<const WordType>}. A constant pointer to a mutable bit shall be obtained through \texttt{const bit\_pointer<WordType>}. A constant pointer to a constant bit shall be obtained through \texttt{const bit\_pointer<const WordType>}.

The return type of the difference between two bit pointers (1.6.2) shall be an implementation-defined signed integer type capable of holding at least as many values as \texttt{ptrdiff\_t}.

\begin{verbatim}
template <class WordType>
class bit_pointer {
  public:
    using word_type = WordType;
    using size_type = see below;
    using difference_type = see below;

    // 1.6.2, types
    bit_pointer() noexcept = default;
    template <class T> constexpr bit_pointer(const bit_pointer<T>& other) noexcept;
    constexpr bit_pointer(nullptr_t) noexcept;
    explicit constexpr bit_pointer(word_type* ptr) noexcept;
    constexpr bit_pointer(word_type* ptr, size_type pos);

    // 1.6.3, constructors
    constexpr bit_pointer& operator=(nullptr_t) noexcept;
    constexpr bit_pointer& operator=(const bit_pointer& other) noexcept;
    template <class T> constexpr bit_pointer& operator=(const bit_pointer<T>& other) noexcept;

    // 1.6.4, assignment
    constexpr bit_pointer& operator++();
    constexpr bit_pointer& operator--();
    constexpr bit_pointer operator++(int);
    constexpr bit_pointer operator--(int);
    constexpr bit_pointer operator+(difference_type n) const;
    constexpr bit_pointer operator-(difference_type n) const;
    constexpr bit_pointer& operator+=(difference_type n);
    constexpr bit_pointer& operator-=(difference_type n);
};
\end{verbatim}

1.6.2 \textit{bit\_pointer member types}  

\textbf{[bit.pointer.types]}

using word_type = WordType;

\textbf{Type}: Refers to the underlying word type that is being provided as a template parameter.

using size_type = see below;

\textbf{Type}: An implementation-defined unsigned integer type capable of holding at least as many values as \texttt{binary\_digits\_v<word\_type>}. Same as \texttt{bit\_value::size\_type (1.4.2)}.

using difference_type = see below;

\textbf{Type}: An implementation-defined signed integer type capable of holding at least as many values as
1.6.3 bit_pointer constructors

bit_pointer() noexcept = default;

Effects: Constructs a default-initialized object of type bit_pointer.

Remarks: Observing (1.6.5) an uninitialized bit pointer, calling member arithmetic operators (1.6.6) on uninitialized bit pointers or calling non-member arithmetic operators (1.6.7) on uninitialized bit pointers leads to an undefined behavior.

template <class T> constexpr bit_pointer(const bit_pointer<T>& other) noexcept;

Effects: Constructs an object of type bit_pointer from another bit pointer other. [Note: This constructor is typically used for implicit conversions of cv-qualified bit pointers. — end note]

Remarks: This constructor shall not participate in overload resolution unless is_convertible_v<T*, word_type*> == true.

customexpr bit_pointer(nullptr_t) noexcept;

Effects: Constructs a null bit pointer.

explicit constexpr bit_pointer(word_type* ptr) noexcept;

Effects: Constructs a pointer to the bit at position 0 of the word pointed to by ptr.

constexpr bit_pointer(word_type* ptr, size_type pos);

Requires: pos < binary_digits_v<word_type>.

Effects: Constructs a pointer to the bit at position pos of the word pointed to by ptr.

Throws: Nothing.

1.6.4 bit_pointer assignment

constexpr bit_pointer& operator=(nullptr_t) noexcept;

Effects: Assigns a null bit pointer to *this.

Returns: *this.

constexpr bit_pointer& operator=(const bit_pointer& other) noexcept;

Effects: Copies the bit pointer other to *this.

Returns: *this.

[Note: The copy assignment operator is not implicitly generated in order to ensure that the pointer itself is changed instead of the value of the bit pointed to by *this. — end note]

template <class T> constexpr bit_pointer& operator=(const bit_pointer<T>& other) noexcept;

Effects: Assigns the bit pointer other to *this.

Returns: *this.

Remarks: This operator shall not participate in overload resolution unless is_convertible_v<T*, word_type*> == true.

1.6.5 bit_pointer observers

explicit constexpr operator bool() const noexcept;

§ 1.6.5
Returns: false if *this is a null bit pointer, true otherwise.

constexpr bit_reference<WordType> operator*() const;
    Requires: static_cast<bool>(*this) == true.
    Returns: A bit_reference (1.5) referencing the bit pointed to by *this.
    Throws: Nothing.

constexpr bit_reference<WordType>* operator->() const;
    Requires: static_cast<bool>(*this) == true.
    Returns: A pointer to a bit_reference (1.5) referencing the bit pointed to by *this.
    Throws: Nothing.

constexpr bit_reference<WordType> operator[](difference_type n) const;
    Requires: static_cast<bool>(*this) == true.
    Returns: A bit_reference (1.5) referencing the n-th bit after (or before for negative n) the bit pointed to by *this according to the arithmetic of bit pointers described in 1.6.1.
    Throws: Nothing.

1.6.6 bit_pointer arithmetic

constexpr bit_pointer& operator++();
    Requires: static_cast<bool>(*this) == true.
    Effects: Increments *this according to the arithmetic of bit pointers described in 1.6.1.
    Returns: *this

constexpr bit_pointer& operator--();
    Requires: static_cast<bool>(*this) == true.
    Effects: Decrements *this according to the arithmetic of bit pointers described in 1.6.1.
    Returns: *this

constexpr bit_pointer operator++(int);
    Requires: static_cast<bool>(*this) == true.
    Effects: Makes a copy of *this, increments *this according to the arithmetic of bit pointers described in 1.6.1, and returns the original copy.
    Returns: A copy of *this made before the increment.

constexpr bit_pointer operator--(int);
    Requires: static_cast<bool>(*this) == true.
    Effects: Makes a copy of *this, decrements *this according to the arithmetic of bit pointers described in 1.6.1, and returns the original copy.
    Returns: A copy of *this made before the decrement.

constexpr bit_pointer operator+(difference_type n) const;
    Requires: static_cast<bool>(*this) == true || n == 0.
    Returns: A bit_pointer pointing to the n-th bit after (or before for negative n) the bit pointed to by

§ 1.6.6
constexpr bit_pointer operator-(difference_type n) const;

Requires: static_cast<bool>(*this) == true || n == 0.
Returns: A bit_pointer pointing to the n-th bit before (or after for negative n) the bit pointed to by *this according to the arithmetic of bit pointers described in 1.6.1.

constexpr bit_pointer& operator+=(difference_type n);

Requires: static_cast<bool>(*this) == true || n == 0.
Returns: *this.

constexpr bit_pointer& operator-=(difference_type n);

Requires: static_cast<bool>(*this) == true || n == 0.
Effects: Increments *this (or decrements for negative n) n times according to the arithmetic of bit pointers described in 1.6.1.
Returns: *this.

1.6.7 bit_pointer non-member operations

template <class T>
constexpr bit_pointer<T> operator+(typename bit_pointer<T>::difference_type n, bit_pointer<T> x);

Requires: static_cast<bool>(x) == true || n == 0.
Returns: x + n.

template <class T, class U>
constexpr common_type_t<typename bit_pointer<T>::difference_type, typename bit_pointer<U>::difference_type> operator-(bit_pointer<T> lhs, bit_pointer<U> rhs);

Requires: lhs->address() - rhs->address() is well-defined.
Returns: If lhs and rhs are both null bit pointers, returns 0. Otherwise, returns the number of bits n such that lhs + n == rhs.

template <class T, class U>
constexpr bool operator==(bit_pointer<T> lhs, bit_pointer<U> rhs) noexcept;

Returns: static_cast<bool>(lhs) == static_cast<bool>(rhs) && (!static_cast<bool>(lhs) || (lhs->address() == rhs->address() && lhs->position() == rhs->position())).

template <class T, class U>
constexpr bool operator!=(bit_pointer<T> lhs, bit_pointer<U> rhs) noexcept;

Returns: static_cast<bool>(lhs) != static_cast<bool>(rhs) || (static_cast<bool>(lhs) && (lhs->address() != rhs->address() || lhs->position() != rhs->position())).

template <class T, class U>
constexpr bool operator<(bit_pointer<T> lhs, bit_pointer<U> rhs) noexcept;

Requires: static_cast<bool>(lhs) == static_cast<bool>(rhs)
Returns: \(\text{static
cast<bool>(lhs) \&\& (lhs->address() < rhs->address()
|| (lhs->address() == rhs->address() \&\& lhs->position() < rhs->position()))}\).

\[
\begin{align*}
\text{template <class T, class U> constexpr bool operator<=(bit_pointer<T> lhs, bit_pointer<U> rhs) noexcept;}
\end{align*}
\]

\[
\begin{align*}
\text{Requires: & static
cast<bool>(lhs) == static
cast<bool>(rhs).}
\end{align*}
\]

\[
\begin{align*}
\text{Returns: & !static
cast<bool>(lhs) || (lhs->address() < rhs->address()
|| (lhs->address() == rhs->address() \&\& lhs->position() <= rhs->position()))}.
\end{align*}
\]

\[
\begin{align*}
\text{template <class T, class U> constexpr bool operator>(bit_pointer<T> lhs, bit_pointer<U> rhs) noexcept;}
\end{align*}
\]

\[
\begin{align*}
\text{Requires: & static
cast<bool>(lhs) == static
cast<bool>(rhs).}
\end{align*}
\]

\[
\begin{align*}
\text{Returns: & !static
cast<bool>(lhs) || (lhs->address() > rhs->address()
|| (lhs->address() == rhs->address() \&\& lhs->position() > rhs->position()))}.
\end{align*}
\]

1.7 Class template bit_iterator

1.7.1 Class template bit_iterator overview

A bit_iterator is an iterator adaptor to iterate over the bits of a range of underlying words. The value-
type (1.7.2) of a bit_iterator is defined as a bit_value (1.4), the reference type (1.7.2) is defined as a bit_reference (1.5) and the pointer type (1.7.2) is defined as a bit_pointer (1.6). [Note: A bit_iterator is typically implemented in terms of a bit position or a mask, and in terms of an underlying iterator. — end note]

The arithmetic of bit iterators (1.7.6) rely on the ordering described in 1.1: a bit iterator it2 is considered to be the next bit iterator of it1 if either of the following is true:

\[
\begin{align*}
\text{(2.1) & it2.base() == it1.base()}
\end{align*}
\]

\[
\begin{align*}
\text{&& it2.position() - it1.position() == 1}
\end{align*}
\]

\[
\begin{align*}
\text{(2.2) & it2.base() == next(it1.base())}
\end{align*}
\]

\[
\begin{align*}
\text{&& binary_digits_v<typename decltype(it1)::word_type> - it1.position() == 1}
\end{align*}
\]

\[
\begin{align*}
\text{&& it2.position() == 0}
\end{align*}
\]

Comparison operators for bit_iterator (1.7.7) rely on the same ordering, first comparing the underlying iterator and then comparing bit positions in case of equality.

The template parameter type Iterator shall be an iterator such that the following types are the same:

\[
\begin{align*}
\text{(3.1) & iterator_traits<Iterator>::value_type}
\end{align*}
\]

\[
\begin{align*}
\text{(3.2) & remove_cv_t<remove_reference_t<typename iterator_traits<Iterator>::reference>>}
\end{align*}
\]

\[
\begin{align*}
\text{(3.3) & remove_cv_t<remove_pointer_t<typename iterator_traits<Iterator>::pointer>>}
\end{align*}
\]

, such that the following types are the same:

\[
\begin{align*}
\text{(3.4) & remove_reference_t<typename iterator_traits<Iterator>::reference>}
\end{align*}
\]

\[
\begin{align*}
\text{(3.5) & remove_pointer_t<typename iterator_traits<Iterator>::pointer>}
\end{align*}
\]

and such that:
can be instantiated, otherwise the program is ill-formed. The member type `word_type` \((1.7.2)\) keeps track of the cv-qualification of the underlying word type. [\textit{Note:} For this reason, the types of `iterator_traits<Iterator>::value_type` and `bit_iterator<Iterator>::word_type` may have different cv-qualifiers. Implementations may use `remove_reference_t<typename iterator_traits<Iterator>::reference>` to propagate cv-qualifiers instead of `iterator_traits<Iterator>::value_type`. — end note]

4 An access to the underlying representation of a `bit_iterator` is provided through the function members base, position and mask \((1.7.5)\).

5 The return type of the difference between two bit iterator \((1.6.2)\) shall be an implementation-defined signed integer type capable of holding at least as many values as `ptrdiff_t`.

```cpp
-template <class Iterator>
class bit_iterator {
 public:
   // 1.7.2, types
   using iterator_type = Iterator;
   using word_type = see below;
   using iterator_category = typename iterator_traits<Iterator>::iterator_category;
   using value_type = bit_value;
   using difference_type = see below;
   using pointer = bit_pointer<word_type>;
   using reference = bit_reference<word_type>;
   using size_type = see below;

   // 1.7.3, constructors
   constexpr bit_iterator();
   template <class T> constexpr bit_iterator(const bit_iterator<T>& other);
   explicit constexpr bit_iterator(iterator_type i);
   constexpr bit_iterator(iterator_type i, size_type pos);

   // 1.7.4, assignment
   template <class T> constexpr bit_iterator& operator=(const bit_iterator<T>& other);

   // 1.7.5, observers
   constexpr reference operator*() const noexcept;
   constexpr pointer operator->() const noexcept;
   constexpr reference operator[](difference_type n) const;
   constexpr iterator_type base() const;
   constexpr size_type position() const noexcept;
   constexpr typename std::remove_cv<word_type>::type mask() const noexcept;

   // 1.7.6, arithmetic
   constexpr bit_iterator& operator++();
   constexpr bit_iterator& operator--();
   constexpr bit_iterator operator++(int);
   constexpr bit_iterator operator--(int);
   constexpr bit_iterator operator+(difference_type n) const;
   constexpr bit_iterator operator-(difference_type n) const;
   constexpr bit_iterator& operator+=(difference_type n);
   constexpr bit_iterator& operator-=(difference_type n);
};
```
1.7.2  bit_iterator member types

using iterator_type = Iterator;

Type: Refers to the Iterator template type parameter that is being adapted.

using word_type = see below;

Type: Refers to the cv-qualified type on which the underlying iterator is iterating, which is equivalent to remove_reference_t<typename iterator_traits<Iterator>::reference> according to 1.7.1.

using iterator_category = typename iterator_traits<Iterator>::iterator_category;

Type: Refers to the same iterator category as the one of the underlying iterator.

using value_type = bit_value;

Type: bit_value.

using difference_type = see below;

Type: An implementation-defined signed integer type capable of holding at least as many values as ptrdiff_t. Same as bit_pointer<word_type>::difference_type (1.6.2).

using pointer = bit_pointer<word_type>;

Type: bit_pointer<word_type>.

using reference = bit_reference<word_type>;

Type: bit_reference<word_type>.

using size_type = see below;

Type: An implementation-defined unsigned integer type capable of holding at least as many values as binary_digits_v<word_type>. Same as bit_value::size_type (1.4.2).

1.7.3  bit_iterator constructors

constexpr bit_iterator();

Effects: Value-initializes the underlying word iterator and the underlying bit position. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type iterator_type.

template <class T> constexpr bit_iterator(const bit_iterator<T>& other);

Requires: is_constructible_v<iterator_type, T> == true

Effects: Constructs an object of type bit_iterator from another bit iterator other, initializing the underlying word iterator from other.base() and initializing the underlying bit position from other.position().

explicit constexpr bit_iterator(iterator_type i);

Effects: Constructs an iterator over the bit at position 0 of the word iterated over by it.

constexpr bit_iterator(iterator_type i, size_type pos);

Requires: pos < binary_digits_v<word_type>.

Effects: Constructs an iterator over the bit at position pos of the word iterated over by it.

Throws: Nothing.
1.7.4  **bit_iterator assignment**  

```cpp
template <class T> constexpr bit_iterator& operator=(const bit_iterator<T>& other);
```

*Requires:* `is_assignable_v<iterator_type, T> == true`

*Effects:* Assigns the bit iterator `other` to `*this`, assigning `other.base()` to the underlying word iterator of `*this` and assigning `other.position()` to the underlying bit position of `*this`.

*Returns:* `*this`.

1.7.5  **bit_iterator observers**

```cpp
constexpr reference operator*() const noexcept;
```

*Returns:* A `bit_reference` (1.5) referencing the bit iterated over by `*this`.

```cpp
constexpr pointer operator->() const noexcept;
```

*Returns:* A `bit_pointer` (1.6) pointing to the bit iterated over by `*this`.

```cpp
constexpr reference operator[](difference_type n) const;
```

*Returns:* A `bit_reference` (1.5) referencing the `n`-th bit after (or before for negative `n`) the bit iterated over by `*this` according to the arithmetic of bit iterators described in 1.7.1.

```cpp
constexpr iterator_type base() const;
```

*Returns:* An iterator over the word containing the bit iterated over by `*this`.

```cpp
constexpr size_type position() const noexcept;
```

*Returns:* The position of the bit iterated over by `*this` within the word containing it.

```cpp
constexpr typename std::remove_cv<word_type>::type mask() const noexcept;
```

*Returns:* A mask of type `std::remove_cv<word_type>::type` whose only set bit is the bit at the position of the bit iterated over by `*this` within the word containing it as in `static_cast<word_type>(1) << position()`.

1.7.6  **bit_iterator arithmetic**

```cpp
constexpr bit_iterator& operator++();
```

*Effects:* Increments `*this` according to the arithmetic of bit iterators described in 1.7.1.

*Returns:* `*this`

```cpp
constexpr bit_iterator& operator--();
```

*Effects:* Decrements `*this` according to the arithmetic of bit iterators described in 1.7.1.

*Returns:* `*this`

```cpp
constexpr bit_iterator operator++(int);
```

*Effects:* Makes a copy of `*this`, increments `*this` according to the arithmetic of bit iterators described in 1.7.1, and returns the original copy.

*Returns:* A copy of `*this` made before the increment.

```cpp
constexpr bit_iterator operator--(int);
```

*Effects:* Makes a copy of `*this`, decrements `*this` according to the arithmetic of bit iterators described in 1.7.1, and returns the original copy.

§ 1.7.6
Returns: A copy of \*this made before the decrement.

\[
\text{constexpr bit_iterator} \quad \text{operator}+(\text{difference_type } n) \quad \text{const;}
\]

Returns: A bit_iterator over the \(n\)-th bit after (or before for negative \(n\)) the bit over which \*this iterates according to the arithmetic of bit iterators described in 1.7.1.

\[
\text{constexpr bit_iterator} \quad \text{operator}-(\text{difference_type } n) \quad \text{const;}
\]

Returns: A bit_iterator over the \(n\)-th bit before (or after for negative \(n\)) the bit over which \*this iterates according to the arithmetic of bit iterators described in 1.7.1.

\[
\text{constexpr bit_iterator} \quad \text{operator}\+=(\text{difference_type } n);
\]

Effects: Increments \*this (or decrements for negative \(n\)) \(n\) times according to the arithmetic of bit iterators described in 1.7.1.

Returns: \*this.

\[
\text{constexpr bit_iterator} \quad \text{operator}\-=(\text{difference_type } n);
\]

Effects: Decrements \*this (or increments for negative \(n\)) \(n\) times according to the arithmetic of bit iterators described in 1.7.1.

Returns: \*this.

### 1.7.7 \texttt{bit\_iterator} non-member operations

- \[\text{template <class T>}
\]
  \[
  \text{constexpr bit\_iterator}<T> \quad \text{operator}+(\text{typename bit\_iterator}<T>::\text{difference\_type } n, \text{const bit\_iterator}<T>& i);
\]

Returns: \(i + n\).

- \[\text{template <class T, class U>}
\]
  \[
  \text{constexpr common\_type_t<}
  \text{typename bit\_iterator}<T>::\text{difference\_type,}
  \text{typename bit\_iterator}<U>::\text{difference\_type}
  \text{>} \quad \text{operator}-(\text{const bit\_iterator}<T>& \text{lhs, const bit\_iterator}<U>& \text{rhs});
\]

Returns: The number of bits \(n\) such that \(lhs + n = rhs\).

- \[\text{template <class T, class U>}
\]
  \[
  \text{constexpr bool} \quad \text{operator}==(<\text{const bit\_iterator}<T>& \text{lhs, const bit\_iterator}<U>& \text{rhs});
\]

Returns: \(lhs.\text{base()} == rhs.\text{base()} \&\& lhs.\text{position()} == rhs.\text{position()}\).

- \[\text{template <class T, class U>}
\]
  \[
  \text{constexpr bool} \quad \text{operator}!=(\text{const bit\_iterator}<T>& \text{lhs, const bit\_iterator}<U>& \text{rhs});
\]

Returns: \(lhs.\text{base()} != rhs.\text{base()} \|\| lhs.\text{position()} != rhs.\text{position()}\).

- \[\text{template <class T, class U>}
\]
  \[
  \text{constexpr bool} \quad \text{operator}<(<\text{const bit\_iterator}<T>& \text{lhs, const bit\_iterator}<U>& \text{rhs});
\]

Returns: \(lhs.\text{base()} < rhs.\text{base()} \|\| (lhs.\text{base()} == rhs.\text{base()}) \&\& lhs.\text{position()} < rhs.\text{position()}\).

- \[\text{template <class T, class U>}
\]
  \[
  \text{constexpr bool} \quad \text{operator}<=(<\text{const bit\_iterator}<T>& \text{lhs, const bit\_iterator}<U>& \text{rhs});
\]

Returns: \(lhs.\text{base()} < rhs.\text{base()} \|\| (lhs.\text{base()} == rhs.\text{base()}) \&\& lhs.\text{position()} <= rhs.\text{position()}\).
template <class T, class U>
constexpr bool operator>(const bit_iterator<T>& lhs, const bit_iterator<U>& rhs);

Returns: \( lhs.base() > rhs.base() \) \| \( lhs.base() == rhs.base() \)
\&\& \( lhs.position() > rhs.position() \).

template <class T, class U>
constexpr bool operator>=(const bit_iterator<T>& lhs, const bit_iterator<U>& rhs);

Returns: \( lhs.base() > rhs.base() \) \| \( lhs.base() == rhs.base() \)
\&\& \( lhs.position() >= rhs.position() \).
Annex A  Comments & remarks  [bit.annex]

1. This annex is not a part of the wording, but comments and remarks on P0237R10.
2. This document corresponds to the updated version of the proposal that has been presented to LEWG during the Toronto meeting, with feedback and comments taken into account. LEWG decided to forward the proposal to LWG in Toronto.
3. History of the proposal includes the original motivating and design review paper P0237R0 (pre-Jacksonville), the wording explorations P0237R1 (pre-Oulu), P0237R2 (post-Oulu), P0237R3 (pre-Issaquah), P0237R4 (post-Issaquah), P0237R5 (pre-Kona), P0237R6 (post-Kona), the formal wording P0237R7 (pre-Toronto), and the version that was approved by LEWG P0237R8 (post-Toronto). The proposal has also been presented at CppCon2016. The Bit Library provides a working implementation [Note: The implementation at a given time t may differ from the proposal by few minor details. — end note] that has been in use at the University of Illinois at Urbana-Champaign since late 2015 with applications in high performance tree data structures, arbitrary precision arithmetic, machine learning and bioinformatics.
4. Throughout the history of the proposal, most design questions have been debated and answered through discussions and polls as reported in the first part of P0237R6. The paper has been presented to LEWG since its first version. The early design has been reviewed by SG14. The paper has been approved by SG6 in Kona.
5. The feedback from users of The Bit Library the University of Illinois at Urbana-Champaign since late 2015 has been very positive, especially regarding to design and performances. The authors have had no problem teaching the library to students, some of whom have contributed to the implementation of bit manipulation algorithms.
6. Long term plans for the standard library based on the bit utilities described in this proposal include high performance overloads of the standard algorithms for bit iterators and a bit container adapter to replace vector<bool> and bitset. Future arbitrary precision numeric types may also benefit from bit utilities to provide an interface to access the underlying representation.
7. The motivations behind bit_value against bool are explained in great depth in P0237R0. Discussions during the Jacksonville meeting favored bit_value against bool. The authors of the paper strongly support the introduction of bit_value in order to avoid some of the misleading behavior users have experienced during the last decades with vector<bool>. Some of the advantages of bit_value over bool can be summarized as follow:

   (7.1) — A bit refers to memory while a bool refers to boolean logic, true, false and conditions, in the same way a byte differs from unsigned char even though both of them have 256 possible values. If a bit and a bool were the same, one could wonder why vector<bool> has been considered to be such a problem. A bit is to a bool what byte is to an unsigned char.

   (7.2) — Using bool instead of bit_value would allow all the implicit conversions of bool, enabling unintuitive behaviors. bit_value provides additional type safety.

   (7.3) — LEWG has given guidance in Oulu to favor the use of member functions for set, reset and flip. The design presented in this proposal allows bit_value and bit_reference to provide a similar interface. bit_value also provides a 2-argument constructor taking a word and a position as arguments, contrarily to bool. Removing bit_value and replacing it by bool would make the writing generic code more difficult.

The name bit_value has been chosen instead of bit to follow the same convention as in bit_reference, bit_pointer and bit_iterator. It also highlights the fact that the class is a wrapper with sizeof(bit_value).
value) >= 1 as any other object in the C++ memory model, the size being expressed as a number of bytes. Feedback from users of The Bit Library regarding bit_value has been very positive. As an additional remark, high-level code often does not use bit_value directly since manipulating bit sequences is achieved through bit_iterator. bit_value only serving as a helper class for bit_iterator::value_type. Since this proposal is targeting a Technical Specification, the Technical Specification could gather more feedback on the use of bit_value instead of bool.

The following points are among those that need to be discussed by LWG:

— The names of bit constants were bikeshedded by LEWG and bit_on/bit_off were suggested. The authors would like the comments of LWG on that.
— Should binary_digits work with char?
— The default constructor of bit_value should initialize it to zero: how should this be specified in the wording?
— The default constructor of bit_pointer should initialize it to a null bit pointer: how should this be specified in the wording?
— How should the bit position be defined?
— The way bit_reference deals with constness (deep const, shallow const) should be checked.
— Is reference convertibility between T and word_type the right test in 1.5.3? Same question for bit values, and bit pointers?
— How the mask member function should produce the mask?
— What should be the lifetime guarantee of the pointer returned by the operator-> of a bit_pointer.
— How should the arithmetic of bit_pointer be specified to avoid out-of-bound scenarios?