Document number: P0237R7
Revises: P0237R6
Date: 2017-06-19
Project: ISO JTC1/SC22/WG21: Programming Language C++
Audience: LEWG
Reply to: Vincent Reverdy and Robert J. Brunner
University of Illinois at Urbana-Champaign
vince.rev@gmail.com

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 [bit]

1.1 General [bit.general]

1 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.

1.2 Header `<bit>` synopsis [bit.syn]

```
namespace std {
  // 1.4, class bit_value
  class bit_value;

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

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

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

  // 1.4.9, bit_value operations
  constexpr bit_value operator~(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;
  constexpr bit_value operator^(bit_value lhs, bit_value rhs) noexcept;

  // 1.5.9, bit_reference swap
  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;

  // 1.6.7, 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 typename std::common_type<
      typename bit_pointer<T>::difference_type,
      typename bit_pointer<U>::difference_type
    >::type operator-(bit_pointer<T> lhs, bit_pointer<U> rhs);

  // 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 typename std::common_type<
```

```
    typename bit_iterator<T>::difference_type,
    typename bit_iterator<U>::difference_type
  >::type operator-(const bit_iterator<T>& lhs, const bit_iterator<U>& rhs);

// 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;

// 1.6.7, bit_pointer comparisons
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;
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;

// 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);

// 1.4.9, 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<<(std::basic_ostream<CharT, Traits>& os,
                                           bit_value x);

// 1.5.9, bit_reference input and output
template <class CharT, class Traits, class T>
  basic_istream<CharT, Traits>& operator>>(std::basic_istream<CharT, Traits>& is,
                                           bit_reference<T>& x);
template <class CharT, class Traits, class T>
  basic_ostream<CharT, Traits>& operator<<(std::basic_ostream<CharT, Traits>& os,
                                           bit_reference<T> x);
```

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

  // 1.4.10, bit_value objects
  static constexpr bit_value bit_zero = bit_value(0U);
  static constexpr bit_value bit_one = bit_value(1U);
}
```

1.3 Helper class `binary_digits` [bit.helper]

1.3.1 Class `binary_digits` overview [bit.helper.overview]

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

1 *Requires:* `UIntType` shall be a (possibly cv-qualified) unsigned integer type. [*Note:* This excludes (possibly cv-qualified) `bool`. — *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 [bit.helper.specializations]

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

1 The specialization of `binary_digits` for (possibly cv-qualified) `byte` makes `byte` a viable word type to hold bits.

1.3.3 Variable template `binary_digits_v` [bit.helper.variable]

```
template <class T> constexpr binary_digits_v = binary_digits<T>::value;
```

1 The variable template `binary_digits_v` provides an access to the `value` member of `binary_digits` for convenience.

1.4 Class `bit_value` [bit.value]

1.4.1 Class `bit_value` overview [bit.value.overview]

1 A `bit_value` emulates the behavior 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 (1.4.7) `set`, `reset` and `flip`. [*Note:* A `bit_value` is typically implemented as a wrapper around `bool`. — *end note*]

2 A `bit_value` is implicitly convertible from a `bit_reference` (1.5), typically to create temporary values from references to bits.

3 To prevent implicit conversions to `bool` and `int` potentially leading to misleading arithmetic behaviors, a `bit_value` is explicitly, and not implicitly, convertible to `bool` (1.4.6).

4 For convenience, two global `bit_value` objects are provided (1.4.10): `bit_zero` and `bit_one`.[1]

```
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> bit_value& operator=(bit_reference<T> ref) noexcept;
  template <class WordType> bit_value& assign(WordType val) noexcept;
  template <class WordType> bit_value& assign(WordType val, size_type pos);

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

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

  // 1.4.7, modifiers
  void set(bool b) noexcept;
  void set() noexcept;
  void reset() noexcept;
  void flip() noexcept;

  // 1.4.8, swap
  void swap(bit_value& other);
  template <class T> void swap(bit_reference<T> other);
};
```

1.4.2 `bit_value` member types [bit.value.types]

```
using size_type = see below;
```

1 *Type:* An implementation-defined unsigned integer type capable of holding at least as many values as `binary_digits_v<word_type>`. Same as `decltype(binary_digits_v<word_type>)` (1.3).

1.4.3 `bit_value` constructors [bit.value.cons]

```
bit_value() noexcept = default;
```

1 *Effects:* Constructs an uninitialized object of type `bit_value`.

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

2 *Effects:* Constructs an object of type `bit_value` from the value of the referenced bit `ref`.

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

1) The full spelling of numbers in the naming avoids the potential ambiguity between `bit_1`, `bit_i`, `bit_I` and `bit_l` in certain fonts.

3 *Requires:* `binary_digits_v<WordType>` shall be defined and shall not be null (1.3).

4 *Effects:* Constructs an object of type `bit_value` from the value of the bit at position `0`.

5 *Remarks:* Contrarily to the more generic constructor that takes an arbitrary position as an argument, this constructor is marked `noexcept`.

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

6 *Requires:* `binary_digits_v<WordType>` shall be defined and shall not be null (1.3).

7 *Requires:* `pos < binary_digits_v<WordType>`.

8 *Effects:* Constructs an object of type `bit_value` from the value of the bit at position `pos`.

1.4.4 `bit_value` assignment [bit.value.assign]

1.4.5 `bit_value` compound assignment [bit.value.cassign]

1.4.6 `bit_value` observers [bit.value.observers]

1.4.7 `bit_value` modifiers [bit.value.modifiers]

1.4.8 `bit_value` swap [bit.value.swap]

1.4.9 `bit_value` non-member operations [bit.value.nonmembers]

1.4.10 `bit_value` objects [bit.value.objects]

1.5 Class template `bit_reference` [bit.reference]

1.5.1 Class template `bit_reference` overview [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). As for `bit_value` (1.4.7), it provides the bit modifier members (1.5.7) `set`, `reset` and `flip`. [*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, and not implicitly, convertible to `bool`.

6 The template parameter type `WordType` shall be an unsigned integer type [*Note:* This does not include `bool`. — *end note*] or a type such that `binary_digits_v<WordType>` is defined and is not null (1.3). A reference to a constant bit shall be obtained through `bit_reference<const WordType>`.

```
template <class WordType>
class bit_reference {
public:
  // 1.5.2, types
  using word_type = WordType;
```

```
  using size_type = see below;

  // 1.5.3, constructors
  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
  bit_reference& operator=(const bit_reference& other) noexcept;
  template <class T> bit_reference& operator=(const bit_reference<T>& other) noexcept;
  bit_reference& operator=(bit_value val) noexcept;
  bit_reference& assign(word_type val) noexcept;
  bit_reference& assign(word_type val, size_type pos);

  // 1.5.5, compound assignment
  bit_reference& operator&=(bit_value other) noexcept;
  bit_reference& operator|=(bit_value other) noexcept;
  bit_reference& operator^=(bit_value other) 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 word_type mask() const noexcept;

  // 1.5.7, modifiers
  void set(bool b) noexcept;
  void set() noexcept;
  void reset() noexcept;
  void flip() noexcept;

  // 1.5.8, swap
  template <class T> void swap(bit_reference<T> other);
  void swap(bit_value& other);
};
```

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 `decltype(binary_digits_v<word_type>)` (1.3).

1.5.3 `bit_reference` constructors [bit.reference.cons]

1.5.4 `bit_reference` assignment [bit.reference.assign]

1.5.5 `bit_reference` compound assignment [bit.reference.cassign]

1.5.6 `bit_reference` observers [bit.reference.observers]

1.5.7 `bit_reference` modifiers [bit.reference.modifiers]

1.5.8 `bit_reference` swap [bit.reference.swap]

1.5.9 `bit_reference` non-member operations [bit.reference.nonmembers]

1.6 Class template `bit_pointer` [bit.pointer]

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

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

2 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).

3 A null bit pointer can be created from a `nullptr` (1.6.3). Deferencing 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.

4 Arithmetic of bit pointers 1.6.6 rely on the following ordering: 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`:

(4.1) — `ptr2->address() - ptr1->address() == 0`
`&& ptr2->position() - ptr1->position() == 1`

(4.2) — `ptr2->address() - ptr1->address() == 1`
`&& binary_digits_v<typename decltype(ptr1)::word_type> - ptr1->position() == 1`
`&& ptr2->position() == 0`

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.

5 The template parameter type `WordType` shall be an unsigned integer type [*Note:* This does not include `bool`. — *end note*] or a type such that `binary_digits_v<WordType>` is defined and is not null (1.3). A pointer to a constant bit shall be obtained through `bit_pointer<const WordType>`. A constant pointer to a mutable bit shall be obtained through `const bit_pointer<WordType>`. A constant pointer to a constant bit shall be obtained through `const bit_pointer<const WordType>`.

6 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 `ptrdiff_t`.

```
template <class WordType>
class bit_pointer {
public:
  // 1.6.2, types
  using word_type = WordType;
  using size_type = see below;
  using difference_type = see below;

  // 1.6.3, constructors
  bit_pointer() noexcept = default;
  template <class T> constexpr bit_pointer(const bit_pointer<T>& other) noexcept;
  constexpr bit_pointer(std::nullptr_t) noexcept;
  explicit constexpr bit_pointer(word_type* ptr) noexcept;
```

```
    constexpr bit_pointer(word_type* ptr, size_type pos);

    // 1.6.4, assignment
    bit_pointer& operator=(std::nullptr_t) noexcept;
    bit_pointer& operator=(const bit_pointer& other) noexcept;
    template <class T> bit_pointer& operator=(const bit_pointer<T>& other) noexcept;

    // 1.6.5, observers
    explicit constexpr operator bool() const noexcept;
    constexpr bit_reference<WordType> operator*() const noexcept;
    constexpr bit_reference<WordType>* operator->() const noexcept;
    constexpr bit_reference<WordType> operator[](difference_type n) const;

    // 1.6.6, arithmetic
    bit_pointer& operator++();
    bit_pointer& operator--();
    bit_pointer operator++(int);
    bit_pointer operator--(int);
    constexpr bit_pointer operator+(difference_type n) const;
    constexpr bit_pointer operator-(difference_type n) const;
    bit_pointer& operator+=(difference_type n);
    bit_pointer& operator-=(difference_type n);
  };
```

1.6.2 `bit_pointer` member types [bit.pointer.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 `decltype(binary_digits_v<word_type>)` (1.3).

```
using difference_type = see below;
```

3 *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.

1.6.3 `bit_pointer` constructors [bit.pointer.cons]

1.6.4 `bit_pointer` assignment [bit.pointer.assign]

1.6.5 `bit_pointer` observers [bit.pointer.observers]

1.6.6 `bit_pointer` arithmetic [bit.pointer.arithmetic]

1.6.7 `bit_pointer` non-member operations [bit.pointer.nonmembers]

1.7 Class template `bit_iterator` [bit.iterator]

1.7.1 Class template `bit_iterator` overview [bit.iterator.overview]

1 A `bit_iterator` is an iterator adaptor to iterate over the bits of a range of underlying values. The `value_type` (1.7.2) of a `bit_iterator` is defined as a `bit_value`, the `reference` type (1.7.2) is defined as a `bit_reference` and the `pointer` type (1.7.2) is defined as a `bit_pointer`. [*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*]

2 Arithmetic of bit iterators 1.7.6 rely on the following ordering: a bit iterator `it2` is considered to be the next bit iterator of `it1` if either of the following is `true`:

(2.1) — `it2.base() == it1.base()`
`&& it2.position() - it1.position() == 1`

(2.2) — `it2.base() == next(it1.base())`
`&& binary_digits_v<typename decltype(it1)::word_type> - it1.position() == 1`
`&& it2.position() == 0`

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.

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

(3.1) — `iterator_traits<Iterator>::value_type`

(3.2) — `remove_cv_t<remove_reference_t<typename iterator_traits<Iterator>::reference>>`

(3.3) — `remove_cv_t<remove_pointer_t<typename iterator_traits<Iterator>::pointer>>`

, such that the following types are the same:

(3.4) — `remove_reference_t<typename iterator_traits<Iterator>::reference>>`

(3.5) — `remove_pointer_t<typename iterator_traits<Iterator>::pointer>>`

and such that:

(3.6) — `bit_reference<remove_reference_t<typename iterator_traits<Iterator>::reference>>`

(3.7) — `bit_pointer<remove_pointer_t<typename iterator_traits<Iterator>::pointer>>`

can be instantiated. [*Note:* This does not include `bool`. — *end note*] The member type `word_type` (1.7.2) keeps track of the cv-qualification of the underlying type. [*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. — *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`.

```
template <class Iterator>
class bit_iterator {
public:
  // 1.7.2, types
  using iterator_type = Iterator;
  using word_type = see below;
  using iterator_category = typename std::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> 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 word_type mask() const noexcept;

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

1.7.2 `bit_iterator` member types [bit.iterator.types]

`using iterator_type = Iterator;`

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

`using word_type = see below;`

2 *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 std::iterator_traits<Iterator>::iterator_category;`

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

`using value_type = bit_value;`

4 *Type:* `bit_value`.

`using difference_type = see below;`

5 *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>;`

6 *Type:* `bit_pointer<word_type>`.

`using reference = bit_reference<word_type>;`

7 *Type:* `bit_reference<word_type>`.

`using size_type = see below;`

8 *Type:* An implementation-defined unsigned integer type capable of holding at least as many values as `binary_digits_v<word_type>`. Same as `decltype(binary_digits_v<word_type>)` (1.3).