Document Number: P0161R0
Date: 2016-02-12
Project: ISO SC22/WG21 C++ Standard, Library Working Group
Reply to: Nathan Myers `<ncm@cantrip.org>`

Bitset Iterators, Masks, and Container Operations

Although the `bitset<>` of C++14 omits many essential bitwise operations, an augmented `bitset<>` would be a near-ideal place to bind specialized bitwise instructions, such as `POPCNT` and `CTZ`, as well as a few of the long-word SIMD operations found on modern platforms. With modern compilers, a round trip of an integer or integer array type through bitset often compiles to exactly zero instructions, so that exposing the available operations through `bitset<>`, rather than as low-level compiler intrinsics, reduces cognitive load on programmers without sacrificing performance.

This paper proposes iterator and set container access, and high-level bindings for the most important operations commonly supported by bitwise instructions on normal and extended register sizes.

Iterators

If `bitset<N>` is a set, we should be able to iterate over its elements and use its iterators in generic algorithms. By definition, an iterator on a set should take on one value for each element in the set. That is, given a bitset value `b`, we should expect `std::distance(b.begin(), b.end()) == b.count()`.

What should `std::bitset<N>::iterator::operator*()` return? A naïve choice would be the numeric index of the bit position represented. In practice, the most useful result is another bitset value `b` such that `b.count() == 1`, usable as a bitmask to apply against other instances. In this view, a bitset is a set, not of the indices of all its bits, or of its 1 bits, but rather of N-bit-long representations of powers of 2 up to 2^{N-1}, inserted into the set by a simple bitwise OR. (This is consistent with its existing constructor from an integer type, which treats the argument as a set of bits, not a bit index.)

This paper proposes a family of iterators, comprising iterator, reverse_iterator, index_iterator, and reverse_index_iterator, and operations on them that work in constant time vs. set cardinality. The additional, index, iterators behave in the naïve way, yielding bit indices as if these were the members of the set.

Iterators proposed here do not behave like the surrogate pointers typical of container iterators. I.e., given an iterator `it`, the expression `*it` is an rvalue that cannot be used to alter the parent bitset, and that does not reflect changes made to the parent bitset subsequent to construction. Iterators also are not invalidated by such changes. Pointer semantics would interfere with efficient implementation and use, and (particularly) with binding to native bitwise instructions. The iterators satisfy requirements on InputIterator types, and (in addition) they allow re-scanning via copies.

Generic Container Operations

We also propose `insert` and `erase` members, both by key and by iterator range (the latter a constant-time operation) for better compatibility with generic code. For the same reason, we propose members `clear()` and `empty()`, and a (constant-time) constructor from iterator ranges.

(We note in passing that, from the standpoint of container conventions, `size()` in bitset is misdefined to report what would be `capacity()` in a proper container, necessitating the extra member `count()` to report the set cardinality that would normally be reported as `size()`. We do not here attempt to repair this historical infelicity; a backward-compatible repair would need to introduce a new type, e.g. `bitset_set<N>`, presumably inheriting from `bitset<N>`. This alternative will taken up if preferred.)

It is not unusual to need to use a `bitset<>` as the key in a container, so we also define operations needed for this, including `op<` *et al.* This is meant to complement (DR1182) `hash` applied to bitsets.

Bindings to Native and SIMD Operations

C++14's `std::bitset<N>` already has member `count()`, which is (in fact) the only portable binding found in C++14 to the assembly-language instruction commonly called `POPCNT`. However, it does not offer quick access to the index position of the lowest or highest bit, typical of `CTZ`-family instructions. It furthermore offers no direct way to get commonly needed related values, such as a mask representing all bits to the right or left of a given index position.

The noted operations are available to a program that can afford to use compiler intrinsics or assembly instructions, or if N is small enough for the bitset to be converted to and from `unsigned long long`. When N is 65 or more, and your code must be portable, no optimal implementation is available. This proposal exposes bindings for these operations on all set capacities.

Modern lexical-scanning techniques achieve radical performance improvements over traditional methods via operations on a collection of bitsets (typically placed in SIMD registers) with one bitset for each bit-position in a char, longitudinally across a character sequence. (I.e., one bitset for the least-significant, or value 1 bit, another for the value 2 bit, and so on up to the 8th, or value 128 bit.) Such optimizations depend, to a large degree, on operations such as proposed here. `std::bitset<N>` offers a standard place to bind native bitwise operations. Subsequent proposals in this vein may bind more of the commonly available operations: most usefully, transposition of a byte sequence into a collection of bitsets.

Some operations defined below are presented with a **Precondition:** spec that washes out common differences between native-instruction versions of the operation (e.g. `CTZ` on a zero value), enabling optimal implementation on a wide variety of hardware. There are complementary, possibly slower, wide-contract versions of each.

Experiments with implementation using Gcc's "vector extensions" have thus far yielded disappointing performance results even for what would seem the best potential uses. Possibly the time needed to marshall values into the extended registers overwhelms benefits from SIMD parallel execution, vs. unrolled operations on regular registers. Meanwhile, the instructions commonly available that operate on extended registers are disappointingly devoid of whole-word operations, most particularly decrement-by-one and shift, which must be implemented word-wise instead.

Notes

Many of the operations proposed here are defined as compositions of other public operations, but are included because they are better done by taking advantage of implementation details, compiler intrinsics, or assembly code. For example, many are specified recursively, but would be implemented wordwise in constant time. The "exposition-only" elements and other code expressions presented are chosen to aid precise definition, not to suggest a production-grade implementation.

Some target instruction sets provide numerous minor variations on these operations, tacking on a complement here, a decrement there. We assume that, where it makes a difference, a peephole optimizer can identify simple compositions and substitute composite instructions, as they do to generate `MODQUOT` and `ROT` instructions, so they need not be exposed at the interface level. (Too often, the composite instructions offer no runtime benefit anyhow. We suppose that their purpose is more for commercial advantage than for utility.) In addition, the members `erase` and `op-=` provide the most commonly useful composite operation.

This proposal does not pretend to cover all useful forms of bitwise data organization. In particular, it does not address data structures with size variable at runtime[3], nor specialty operations accelerated in hardware in only a minority of target machines. These latter operations can be considered where merited.

We do not consider it limiting to bind these operations to library class templates, rather than to native machine words as in [2]. Compilers are lately very good at seeing past compile-time fixed-size struct organization of rvalues to the arrays and native values they enwrap, so that, e.g., a std::bitset may occupy the same register as an integer (or integer array) value it converts to or from.

In contrast to typical function interfaces elsewhere in the standard library, these functions take their arguments and return results *by value* wherever possible. This approach reserves a maximum of freedom for optimizers. The apparently-implied copy operations are routinely elided by compilers when instantiating the mostly inline functions.

Note further that most of the operations proposed are declared constexpr, making them useful, at compile time, to construct bitwise structures of any size.

Proposal

Declare `constexpr` all members of class bitset that do not involve iostreams.

Add to class `bitset` the following.

Class bitset

```
template <size_t N>
class bitset {

    [ ... ]

    using key_type = bitset;
    using value_type = bitset;

    constexpr bitset(iterator b, iterator e);
    constexpr bitset(reverse_iterator b, reverse_iterator e);
    constexpr bitset(index_iterator b, index_iterator e);
    constexpr bitset(reverse_index_iterator b, reverse_index_iterator e);

    constexpr void insert(bitset s);
    constexpr void insert(iterator b, iterator e);
    constexpr void insert(reverse_iterator b, reverse_iterator e);
    constexpr void insert(index_iterator b, index_iterator e);
    constexpr void insert(
        reverse_index_iterator b, reverse_index_iterator e);
    constexpr void erase(bitset s);
    constexpr void erase(iterator b, iterator e);
    constexpr void erase(reverse_iterator b, reverse_iterator e);
    constexpr void erase(index_iterator b, index_iterator e);
    constexpr void erase(
        reverse_index_iterator b, reverse_index_iterator e);

    constexpr void clear();
    constexpr bool empty() const;

    constexpr bool operator<(bitset s) const;
    constexpr bool operator<=(bitset s) const;
    constexpr bool operator>(bitset s) const;
    constexpr bool operator>=(bitset s) const;

    constexpr bitset& operator-=(bitset s);

    constexpr size_t low_bit_position() const;
    constexpr bitset low_bit() const;
    constexpr bitset low_mask() const;
    constexpr bitset low_mask_not() const;

    constexpr size_t high_bit_position() const;
    constexpr bitset high_bit() const;
    constexpr bitset high_mask() const;
    constexpr bitset high_mask_not() const;

    constexpr bitset range_mask(size_t n) const;

    class iterator
        public iterator<input_iterator_tag, bitset, size_t, void, void>
    {
         bitset place_; // exposition only
         explicit constexpr iterator(bitset s)   // exposition only
             : place_(s) {}
      public:
         constexpr iterator();
         constexpr bitset operator*() const;
         constexpr iterator& operator++();
         constexpr iterator operator++(int);
         constexpr bool operator==(iterator other) const;
         constexpr bool operator!=(iterator other) const;
         constexpr iterator operator-(iterator other) const;
    };
    constexpr iterator begin() const;
    constexpr iterator end() const;
    using const_iterator = iterator;
    constexpr iterator cbegin() const;
    constexpr iterator cend() const;

    class reverse_iterator : public iterator
        public iterator<input_iterator_tag, bitset, size_t, void, void>
    {
         bitset place_; // exposition only
         explicit constexpr reverse_iterator(bitset s)
            : place_(s) {}  // exposition only
      public:
         constexpr reverse_iterator();
         constexpr bitset operator*() const;
         constexpr reverse_iterator& operator++();
         constexpr reverse_iterator operator++(int);
         constexpr bool operator==(reverse_iterator other) const;
         constexpr bool operator!=(reverse_iterator other) const;
```

```
            constexpr reverse_iterator operator-(iterator reverse_other) const;
        };
        using const_reverse_iterator = reverse_iterator;
        constexpr reverse_iterator rbegin() const;
        constexpr reverse_iterator rend() const;
        constexpr reverse_iterator crbegin() const;
        constexpr reverse_iterator crend() const;

        class index_iterator:
            public iterator<input_iterator_tag, bitset, size_t, void, void>
        {
             bitset place_; // exposition only
             explicit constexpr index_iterator(bitset s)
                : place_(s) {}  // exposition only
          public:
             constexpr index_iterator();
             constexpr size_t operator*() const;
             constexpr index_iterator& operator++();
             constexpr index_iterator operator++(int);
             constexpr bool operator==(index_iterator other) const;
             constexpr bool operator!=(index_iterator other) const;
             constexpr index_iterator operator-(iterator index_other) const;
        };
        using const_index_iterator = index_iterator;
        constexpr index_iterator ibegin() const;
        constexpr index_iterator iend() const;
        constexpr index_iterator cibegin() const;
        constexpr index_iterator ciend() const;

        class reverse_index_iterator:
            public iterator<input_iterator_tag, bitset, size_t, void, void>
        {
             bitset place_; // exposition only
             explicit constexpr reverse_index_iterator(bitset s)
                : place_(s) {}  // exposition only
          public:
             constexpr reverse_index_iterator();
             constexpr size_t operator*() const;
             constexpr reverse_index_iterator& operator++();
             constexpr reverse_index_iterator operator++(int);
             constexpr bool operator==(reverse_index_iterator other) const;
             constexpr bool operator!=(reverse_index_iterator other) const;
             constexpr reverse_index_iterator operator-(
                 iterator reverse_index_other) const;
        };
        using const_reverse_index_iterator = reverse_index_iterator;
        constexpr reverse_index_iterator ribegin() const;
        constexpr reverse_index_iterator riend() const;
        constexpr reverse_index_iterator cribegin() const;
        constexpr reverse_index_iterator criend() const;
    };
```

In addition, add constexpr overloads of a free function template from `<iterator>`, in support of generic programming:

```
template <size_t N>
size_t distance<bitset<N>::iterator>(bitset<N>::iterator a, bitset<N>::iterator<N> b);
template <size_t N>
size_t distance<bitset<N>::reverse_iterator>(
    bitset<N>::reverse_iterator a, bitset<N>::reverse_iterator<N> b);
template <size_t N>
```

```
size_t distance<bitset<N>::index_iterator>(
    bitset<N>::index_iterator a, bitset<N>::index_iterator<N> b);
template <size_t N>
size_t distance<bitset<N>::reverse_index_iterator>(
    bitset<N>::reverse_index_iterator a, bitset<N>::reverse_index_iterator<N> b);
```

Returns: `(b.place_ & ~e.place_).count()`

And, at namespace scope, the constexpr overload:

```
template <size_t N>
bitset<N> operator-(bitset<N> a, bitset<N> b);
```

Returns: `a -= b`

Bitset Members

```
bitset(iterator b, iterator e);
bitset(reverse_iterator b, reverse_iterator e);
bitset(index_iterator b, index_iterator e);
bitset(reverse_index_iterator b, reverse_index_iterator e);
```

Returns: `b.place_ & ~e.place_`

```
void insert(bitset e);
```

Effect: `operator|=(e)`

```
void insert(iterator b, iterator e);
void insert(reverse_iterator b, reverse_iterator e);
void insert(index_iterator b, index_iterator e);
void insert(reverse_index_iterator b, reverse_index_iterator e);
```

Returns: `operator|=(b.place_ & ~e.place_);`

```
void erase(bitset e);
```

Effect: `operator&=(~e)`

```
void erase(iterator b, iterator e);
void erase(reverse_iterator b, reverse_iterator e);
void erase(index_iterator b, index_iterator e);
void erase(reverse_index_iterator b, reverse_index_iterator e);
```

Effect: `operator&=(~(b.place_ & ~e.place_))`

```
void clear();
```

Effect: `reset()`

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

Returns: `none()`

`bool operator<(bitset s) const;`

> **Returns**: if `*this == s`, `false`; else, if `none()`, `true`; else, if `s.none()`, `false`; else, if `high_bit() == s.high_bit()`, `(*this ^ high_bit()) < (s ^ s.high_bit())`; else, `high_bit_position() < s.high_bit_position()`.

`bool operator<=(bitset s) const;`

> **Returns**: `!(s < *this)`

`bool operator>(bitset s) const;

> **Returns**: `s < *this`

`bool operator>=(bitset s) const;`

> **Returns**: `!(*this < s)`

`bitset& operator-=(bitset s);`

> **Returns**: *this &= ~s;

`bitset operator-(bitset s) const;`

> **Returns**: `*this & ~s`

`size_t low_bit_position() const;`

> **Precondition**: `any() == true`
> **Returns**: `test(0) ? 0 : 1 + low_bit_position(*this >> 1)` *[Note: this is also the number of zero bits to the right of the low bit. —end note]*

`bitset low_bit() const;`

> **Precondition**: `any() == true`
> **Returns**: `bitset{}.set(low_bit_position())`

`bitset low_mask() const;`

> **Precondition**: `any() == true`
> **Returns**: `test(0) ? bitset{} : (low_mask(*this >> 1) << 1).set(0)`

`bitset low_mask_not() const;`

> **Returns**: `none() ? ~bitset{} : ~low_mask()`

`size_t high_bit_position() const;`

> **Precondition**: `any() == true`
> **Returns**: `test(N-1) ? N-1 : high_bit_position(*this << 1) - 1`

`bitset high_bit() const;`

> **Precondition**: `any() == true`

Returns: `bitset{}.set(high_bit_position())` *[Note: This is the floor of $\log_2$ of the bitset viewed as an integer type —end note]*

```
bitset high_mask() const;
```

Precondition: `any() == true`
Returns: `test(N-1) ? bitset{} : (high_mask(*this << 1) >> 1).set(N-1)`

```
bitset high_mask_not() const;
```

Returns: `none() ? ~bitset{} : ~high_mask()`

```
bitset range_mask(size_t n) const;
```

Returns: `~high_mask_not() & bitset{}.set(n).low_mask_not()`

```
iterator begin() const;
iterator cbegin() const;
```

Returns: [as if] `iterator(*this)`

```
iterator end() const;
iterator cend() const;
```

Returns: `iterator{}`

```
reverse_iterator rbegin() const;
reverse_iterator crbegin() const;
```

Returns: [as if] `reverse_iterator(*this)`

```
reverse_iterator rend() const;
reverse_iterator crend() const;
```

Returns: `reverse_iterator{}`

```
index_iterator ibegin() const;
index_iterator cibegin() const;
```

Returns: [as if] `index_iterator(*this)`

```
index_iterator iend() const;
index_iterator crend() const;
```

Returns: `reverse_iterator{}`

```
reverse_index_iterator ribegin() const;
reverse_index_iterator cribegin() const;
```

Returns: [as if] `reverse_index_iterator(*this)`

```
reverse_index_iterator riend() const;
reverse_index_iterator criend() const;
```

Returns: `reverse_index_iterator{}`

Iterator Members

```
iterator::iterator();
```

Effect: [as if] `place_ = bitset{}`
Remark: While this presents as an InputIterator, multiple passes are allowed using additional copies of an iterator.

```
bitset iterator::operator*() const;
```

Precondition: `*this != iterator{}`
Returns: `place_.low_bit()`

```
iterator& iterator::operator++();
```

Precondition: `*this != iterator{}`
Effect: `place_ &= ~place_.low_bit()`
Returns: `*this`

```
iterator iterator::operator++(int);
```

Precondition: `*this != iterator{}`
Effect: `place_ &= ~place_.low_bit()`
Returns: a copy of the previous value of `*this`

```
bool iterator::operator==(iterator other) const;
```

Returns: `place_ == other.place_`

```
bool iterator::operator!=(iterator other) const;
```

Returns: `place_ != other.place_`

```
bool iterator::operator-(reverse_iterator other) const;
```

Returns: `iterator(this->place_ & ~other.place_)`

Reverse Iterator Members

```
reverse_iterator::reverse_iterator();
```

Effect: [as if] `place_ = bitset{}`
Remark: While this presents as an InputIterator, multiple passes are allowed using additional copies of an iterator.

```
bitset reverse_iterator::operator*() const;
```

Precondition: `*this != reverse_iterator{}`
Returns: `place_.high_bit()`

```
reverse_iterator& reverse_iterator::operator++();
```

Precondition: `*this != reverse_iterator{}`
Effect: `place_ &= ~place_.high_bit()`
Returns: `*this`

```
reverse_iterator reverse_iterator::operator++(int);
```

Precondition: `*this != reverse_iterator{}`
Effect: `place_ &= ~place_.high_bit()`
Returns: a copy of the previous value of `*this`

```
bool reverse_iterator::operator==(reverse_iterator other) const;
```

Returns: `place_ == other.place_`

```
bool reverse_iterator::operator!=(reverse_iterator other) const;
```

Returns: `place_ != other.place_`

```
bool index_iterator::operator-(reverse_iterator other) const;
```

Returns: `reverse_iterator(this->place_ & ~other.place_)`

Index Iterator Members

```
index_iterator::index_iterator();
```

Effect: [as if] `place_ = bitset{}`
Remark: While this presents as an InputIterator, multiple passes are allowed using additional copies of an iterator.

```
size_t index_iterator::operator*() const;
```

Precondition: `*this != index_iterator{}`
Returns: `place_.low_bit_position()`

```
index_iterator& index_iterator::operator++();
```

Precondition: `*this != index_iterator{}`
Effect: `place_ &= ~place_.low_bit()`
Returns: `*this`

```
index_iterator index_iterator::operator++(int);
```

Precondition: `*this != index_iterator{}`
Effect: `place_ &= ~place_.low_bit()`
Returns: a copy of the previous value of `*this`

```
bool index_iterator::operator==(index_iterator other) const;
```

Returns: `place_ == other.place_`

```
bool index_iterator::operator!=(index_iterator other) const;
```

Returns: `place_ != other.place_`

`bool index_iterator::operator-(index_iterator other) const;`

> **Returns**: `_index_iterator(this->place_ & ~other.place_)`

Reverse Index Iterator Members

`reverse_index_iterator::reverse_index_iterator();`

> **Effect**: [as if] `place_ = bitset{}`
> **Remark**: While this presents as an InputIterator, multiple passes are allowed using additional copies of an iterator.

`size_t reverse_index_iterator::operator*() const;`

> **Precondition**: `*this != reverse_index_iterator{}`
> **Returns**: `place_.high_bit_position()`

`reverse_index_iterator& reverse_index_iterator::operator++();`

> **Precondition**: `*this != reverse_index_iterator{}`
> **Effect**: `place_ &= ~place_.high_bit()`
> **Returns**: `*this`

`reverse_index_iterator reverse_index_iterator::operator++(int);`

> **Precondition**: `*this != reverse_index_iterator{}`
> **Effect**: `place_ &= ~place_.high_bit()`
> **Returns**: a copy of the previous value of `*this`

`bool reverse_index_iterator::operator==(reverse_index_iterator other) const;`

> **Returns**: `place_ == other.place_`

`bool reverse_index_iterator::operator!=(reverse_index_iterator other) const;`

> **Returns**: `place_ != other.place_`

`bool reverse_index_iterator::operator-(reverse_index_iterator other) const;`

> **Returns**: `reverse_index_iterator(this->place_ & ~other.place_)`

Open questions

1. Given the historical baggage of members `size()` and `count()`, and unfortunate implementation / ABI choices that make a `bitset<8>` unnecessarily occupy 8 bytes with a mixed-endian memory layout, should we instead introduce a wholly new type `bitset_set<>` resolving `bitset`'s infelicities?

2. Is there any value in defining an end_iterator type derived from iterator, with its own operators == and != against iterator, that can be implemented more efficiently? Or does inline definition of end() provide a compiler all the information it needs to do just as well?

3. What important low-level operations are not represented yet? (I have omitted those that

should be trivially folded from constituent operations by a peephole optimizer.)

   a. Do we need composed forms, e.g. `b |= m << 100` and `b &= ~(m << 100)`

4. What would a bitwise transpose from a sequence of char/short/int to a commensurate collection of bitsets look like? (e.g. eight bitsets from a `string_view`)

5. What would integration with ranges, range iterators, and/or proxy iterators involve? Is this actually desirable; i.e., can we get it with exactly zero runtime cost? Does it necessarily impose awkward layout restrictions?

6. How should conversion to/from an array of integers commensurate with N look? This is important for compatibility with other encapsulated interpretations of multiple words, to avoid redundant copying and enable a bitset to live in the same register as, e.g., `std::array` or a multiprecision integer type.

References

1. P0125R0 “std::bitset inclusion test methods”, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0125r0.html
2. N3864 “A constexpr bitwise operations library for C++”, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3864.html
3. Howard Hinnant, “On `vector<bool>`”: http://howardhinnant.github.io/onvectorbool.html
4. libc++ `bit_reference` implementation: https://github.com/llvm-mirror/libcxx/blob/master/include/__bit_reference
5. Gcc “vector intrinsics” http://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html
6. Clang “extended vectors” http://clang.llvm.org/docs/LanguageExtensions.html#vectors-and-extended-vectors
7. Portable-esque “SSE intrinsics” http://www.linuxjournal.com/content/introduction-gcc-compiler-intrinsics-vector-processing
8. MSVC SSE intrinsics (cf. (7)) https://msdn.microsoft.com/en-us/library/y0dh78ez(VS.80).aspx