Making `std::vector` constexpr

Document #:	P1004R1
Date:	2018-10-07
Project:	Programming Language C++
Audience:	LWG

Reply-to: Louis Dionne <ldionne@apple.com>

1 Revision history

- R0 – Initial draft
- R1 –
  - Per LEWG guidance in RAP, specify that `std::vector`'s iterators are constexpr iterators, as defined in [P0858R0].
  - Remove an easter egg from the wording – I don't mess with LWG.
  - Other minor fixes from LWG Batavia meeting.

2 Abstract

`std::vector` is not currently `constexpr` friendly. With the loosening of requirements on `constexpr` in [P0784R1] and related papers, we can now make `std::vector` `constexpr`, and we should in order to support the `constexpr` reflection effort (and other evident use cases).

3 Encountered issues

We surveyed the implementation of `std::vector` in libc++ and noted the following issues:

- ASAN and debug annotations (like iterator invalidation checks) can't work in `constexpr`.
- Assertions won't work in `constexpr`.
- `pointer_traits<T*>::pointer_to` is used but is not currently `constexpr`.
- Try-catch blocks are used in some places (e.g. `std::vector::insert`), but those can't appear in `constexpr`.
- Note that making `std::swap` `constexpr` is not a problem since the resolution of [P0859R0], according to Richard Smith.

Assertion and ASAN annotations can be handled by having a mechanism to detect when a function is evaluated as part of a constant expression, as proposed in [P0595R0].

`std::pointer_traits` can be made `constexpr` in the cases we care about; this is handled by P1006, which should be published in the same mailing as this paper.

Try-catch blocks could be allowed inside `constexpr`; this is handled by P1002, which should be published in the same mailing as this paper.

4 Proposed wording

This wording is based on the working draft [N4727]. We basically mark all the member and non-member functions of `std::vector` `constexpr`.

Change in [**vector.syn**] **21.3.6**:

```
#include <initializer_list>

namespace std {
  // 21.3.11, class template vector
  template<class T, class Allocator = allocator<T>> class vector;

  template<class T, class Allocator>
    constexpr bool operator==(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
  template<class T, class Allocator>
    constexpr bool operator!=(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
  template<class T, class Allocator>
    constexpr bool operator< (const vector<T, Allocator>& x, const vector<T, Allocator>& y);
  template<class T, class Allocator>
    constexpr bool operator> (const vector<T, Allocator>& x, const vector<T, Allocator>& y);
  template<class T, class Allocator>
    constexpr bool operator>=(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
  template<class T, class Allocator>
    constexpr bool operator<=(const vector<T, Allocator>& x, const vector<T, Allocator>& y);

  template<class T, class Allocator>
    constexpr void swap(vector<T, Allocator>& x, vector<T, Allocator>& y)
      noexcept(noexcept(x.swap(y)));

  [...]
}
```

Add after [**vector.overview**] **21.3.11.1/2**:

> The types `iterator` and `const_iterator` meet the constexpr iterator requirements ([iterator.requirements.general]).

Change in [**vector.overview**] **21.3.11.1**:

```
namespace std {
  template<class T, class Allocator = allocator<T>>
```

```
class vector {
public:
  // types
  using value_type             = T;
  using allocator_type         = Allocator;
  using pointer                = typename allocator_traits<Allocator>::pointer;
  using const_pointer          = typename allocator_traits<Allocator>::const_pointer;
  using reference              = value_type&;
  using const_reference        = const value_type&;
  using size_type              = implementation-defined; // see 21.2
  using difference_type        = implementation-defined; // see 21.2
  using iterator               = implementation-defined; // see 21.2
  using const_iterator         = implementation-defined; // see 21.2
  using reverse_iterator       = std::reverse_iterator<iterator>;
  using const_reverse_iterator = std::reverse_iterator<const_iterator>;

  // 21.3.11.2, construct/copy/destroy
  constexpr vector() noexcept(noexcept(Allocator())) : vector(Allocator()) { }
  constexpr explicit vector(const Allocator&) noexcept;
  constexpr explicit vector(size_type n, const Allocator& = Allocator());
  constexpr vector(size_type n, const T& value, const Allocator& = Allocator());
  template<class InputIterator>
    constexpr vector(InputIterator first, InputIterator last, const Allocator& = Allocator());
  constexpr vector(const vector& x);
  constexpr vector(vector&&) noexcept;
  constexpr vector(const vector&, const Allocator&);
  constexpr vector(vector&&, const Allocator&);
  constexpr vector(initializer_list<T>, const Allocator& = Allocator());
  constexpr ~vector();
  constexpr vector& operator=(const vector& x);
  constexpr vector& operator=(vector&& x)
    noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
             allocator_traits<Allocator>::is_always_equal::value);
  constexpr vector& operator=(initializer_list<T>);
  template<class InputIterator>
    constexpr void assign(InputIterator first, InputIterator last);
  constexpr void assign(size_type n, const T& u);
  constexpr void assign(initializer_list<T>);
  constexpr allocator_type get_allocator() const noexcept;

  // iterators
  constexpr iterator               begin() noexcept;
  constexpr const_iterator         begin() const noexcept;
  constexpr iterator               end() noexcept;
  constexpr const_iterator         end() const noexcept;
  constexpr reverse_iterator       rbegin() noexcept;
  constexpr const_reverse_iterator rbegin() const noexcept;
  constexpr reverse_iterator       rend() noexcept;
  constexpr const_reverse_iterator rend() const noexcept;

  constexpr const_iterator         cbegin() const noexcept;
```

```
  constexpr const_iterator         cend() const noexcept;
  constexpr const_reverse_iterator crbegin() const noexcept;
  constexpr const_reverse_iterator crend() const noexcept;

  // 21.3.11.3, capacity
  constexpr [[nodiscard]] bool empty() const noexcept;
  constexpr size_type size() const noexcept;
  constexpr size_type max_size() const noexcept;
  constexpr size_type capacity() const noexcept;
  constexpr void      resize(size_type sz);
  constexpr void      resize(size_type sz, const T& c);
  constexpr void      reserve(size_type n);
  constexpr void      shrink_to_fit();

  // element access
  constexpr reference       operator[](size_type n);
  constexpr const_reference operator[](size_type n) const;
  constexpr const_reference at(size_type n) const;
  constexpr reference       at(size_type n);
  constexpr reference       front();
  constexpr const_reference front() const;
  constexpr reference       back();
  constexpr const_reference back() const;

  // 21.3.11.4, data access
  constexpr T*       data() noexcept;
  constexpr const T* data() const noexcept;

  // 21.3.11.5, modifiers
  template<class... Args> constexpr reference emplace_back(Args&&... args);
  constexpr void push_back(const T& x);
  constexpr void push_back(T&& x);
  constexpr void pop_back();

  template<class... Args> constexpr iterator emplace(const_iterator position, Args&&... args);
  constexpr iterator insert(const_iterator position, const T& x);
  constexpr iterator insert(const_iterator position, T&& x);
  constexpr iterator insert(const_iterator position, size_type n, const T& x);
  template<class InputIterator>
  constexpr iterator insert(const_iterator position, InputIterator first, InputIterator last);
  constexpr iterator insert(const_iterator position, initializer_list<T> il);
  constexpr iterator erase(const_iterator position);
  constexpr iterator erase(const_iterator first, const_iterator last);
  constexpr void     swap(vector&)
    noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
             allocator_traits<Allocator>::is_always_equal::value);
  constexpr void     clear() noexcept;
};

template<class InputIterator,
         class Allocator = allocator<iter-value-type<InputIterator>>>
```

```
    vector(InputIterator, InputIterator, Allocator = Allocator())
      -> vector<iter-value-type<InputIterator>, Allocator>;

  // swap
  template<class T, class Allocator>
    constexpr void swap(vector<T, Allocator>& x, vector<T, Allocator>& y)
      noexcept(noexcept(x.swap(y)));
}
```

Change in [**vector.cons**] **21.3.11.2**:

```
constexpr explicit vector(const Allocator&);
```

[...]

```
constexpr explicit vector(size_type n, const Allocator& = Allocator());
```

[...]

```
constexpr vector(size_type n, const T& value,
                 const Allocator& = Allocator());
```

[...]

```
template<class InputIterator>
  constexpr vector(InputIterator first, InputIterator last,
                   const Allocator& = Allocator());
```

[...]

Change in [**vector.capacity**] **21.3.11.3**:

```
constexpr size_type capacity() const noexcept;
```

[...]

```
constexpr void reserve(size_type n);
```

[...]

```
constexpr void shrink_to_fit();
```

[...]

```
constexpr void swap(vector& x)
  noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
           allocator_traits<Allocator>::is_always_equal::value);
```

[...]

```
constexpr void resize(size_type sz);
```

[...]

```
constexpr void resize(size_type sz, const T& c);
```

[...]

Change in [**vector.data**] **21.3.11.4**:

```
constexpr T*         data() noexcept;
constexpr const T*   data() const noexcept;
```

Change in [**vector.modifiers**] **21.3.11.5**:

```
constexpr iterator insert(const_iterator position, const T& x);
constexpr iterator insert(const_iterator position, T&& x);
constexpr iterator insert(const_iterator position, size_type n, const T& x);
template<class InputIterator>
  constexpr iterator insert(const_iterator position, InputIterator first, InputIterator last);
constexpr iterator insert(const_iterator position, initializer_list<T>);

template<class... Args> constexpr reference emplace_back(Args&&... args);
template<class... Args> constexpr iterator emplace(const_iterator position, Args&&... args);
constexpr void push_back(const T& x);
constexpr void push_back(T&& x);
```

[...]

```
constexpr iterator erase(const_iterator position);
constexpr iterator erase(const_iterator first, const_iterator last);
constexpr void pop_back();
```

Change in [**vector.special**] **21.3.11.6**:

```
template<class T, class Allocator>
  constexpr void swap(vector<T, Allocator>& x, vector<T, Allocator>& y)
    noexcept(noexcept(x.swap(y)));
```

Change in [**vector.bool**] **21.3.12/1**:

To optimize space allocation, a specialization of vector for `bool` elements is provided:

```
namespace std {
  template<class Allocator>
  class vector<bool, Allocator> {
  public:
    // types
    using value_type             = bool;
    using allocator_type         = Allocator;
    using pointer                = implementation-defined;
    using const_pointer          = implementation-defined;
    using const_reference        = bool;
    using size_type              = implementation-defined; // see 21.2
    using difference_type        = implementation-defined; // see 21.2
    using iterator               = implementation-defined; // see 21.2
    using const_iterator         = implementation-defined; // see 21.2
    using reverse_iterator       = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;

    // bit reference
    class reference {
```

```
  friend class vector;
  constexpr reference() noexcept;
public:
  constexpr ~reference();
  constexpr reference(const reference&) = default;
  constexpr operator bool() const noexcept;
  constexpr reference& operator=(const bool x) noexcept;
  constexpr reference& operator=(const reference& x) noexcept;
  constexpr void flip() noexcept;     // flips the bit
};

// construct/copy/destroy
constexpr vector() : vector(Allocator()) { }
constexpr explicit vector(const Allocator&);
constexpr explicit vector(size_type n, const Allocator& = Allocator());
constexpr vector(size_type n, const bool& value, const Allocator& = Allocator());
template<class InputIterator>
  constexpr vector(InputIterator first, InputIterator last, const Allocator& = Allocator());
constexpr vector(const vector& x);
constexpr vector(vector&& x);
constexpr vector(const vector&, const Allocator&);
constexpr vector(vector&&, const Allocator&);
constexpr vector(initializer_list<bool>, const Allocator& = Allocator()));
constexpr ~vector();
constexpr vector& operator=(const vector& x);
constexpr vector& operator=(vector&& x);
constexpr vector& operator=(initializer_list<bool>);
template<class InputIterator>
  constexpr void assign(InputIterator first, InputIterator last);
constexpr void assign(size_type n, const bool& t);
constexpr void assign(initializer_list<bool>);
constexpr allocator_type get_allocator() const noexcept;

// iterators
constexpr iterator               begin() noexcept;
constexpr const_iterator         begin() const noexcept;
constexpr iterator               end() noexcept;
constexpr const_iterator         end() const noexcept;
constexpr reverse_iterator       rbegin() noexcept;
constexpr const_reverse_iterator rbegin() const noexcept;
constexpr reverse_iterator       rend() noexcept;
constexpr const_reverse_iterator rend() const noexcept;

constexpr const_iterator         cbegin() const noexcept;
constexpr const_iterator         cend() const noexcept;
constexpr const_reverse_iterator crbegin() const noexcept;
constexpr const_reverse_iterator crend() const noexcept;

// capacity
constexpr [[nodiscard]] bool empty() const noexcept;
constexpr size_type size() const noexcept;
```

```
    constexpr size_type max_size() const noexcept;
    constexpr size_type capacity() const noexcept;
    constexpr void      resize(size_type sz, bool c = false);
    constexpr void      reserve(size_type n);
    constexpr void      shrink_to_fit();

    // element access
    constexpr reference       operator[](size_type n);
    constexpr const_reference operator[](size_type n) const;
    constexpr const_reference at(size_type n) const;
    constexpr reference       at(size_type n);
    constexpr reference       front();
    constexpr const_reference front() const;
    constexpr reference       back();
    constexpr const_reference back() const;

    // modifiers
    template<class... Args> constexpr reference emplace_back(Args&&... args);
    constexpr void push_back(const bool& x);
    constexpr void pop_back();
    template<class... Args> constexpr iterator emplace(const_iterator position, Args&&... args);
    constexpr iterator insert(const_iterator position, const bool& x);
    constexpr iterator insert(const_iterator position, size_type n, const bool& x);
    template<class InputIterator>
      constexpr iterator insert(const_iterator position, InputIterator first, InputIterator last
    constexpr iterator insert(const_iterator position, initializer_list<bool> il);

    constexpr iterator erase(const_iterator position);
    constexpr iterator erase(const_iterator first, const_iterator last);
    constexpr void swap(vector&);
    constexpr static void swap(reference x, reference y) noexcept;
    constexpr void flip() noexcept;       // flips all bits
    constexpr void clear() noexcept;
  };
}
```

Change in **[vector.bool] 21.3.12/4**:

```
constexpr void flip() noexcept;
```

Change in **[vector.bool] 21.3.12/5**:

```
constexpr static void swap(reference x, reference y) noexcept;
```

5 References

[N4727] Richard Smith, *Working Draft, Standard for Programming Language C++* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4727.pdf

[P0784R1] Multiple authors, *Standard containers and constexpr*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0784r1.html

[P0859R0] Richard Smith, *Core Issue 1581: When are constexpr member functions defined?*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0859r0.html

[P0595R0] David Vandevoorde, *The constexpr Operator*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0595r0.html

[P0858R0] Antony Polukhin, *Constexpr iterator requirements*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0858r0.html