Making `std::deque` constexpr

Document #:	P0000
Date:	2019-08-11
Project:	Programming Language C++
Audience:	LEWGI

Reply-to: Alexander Zaitsev <zamazan4ik@tut.by, zamazan4ik@gmail.com>

1 Revision history

- R0 – Initial draft

2 Abstract

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

3 Motivation

`std::deque` is not so widely-used standard container as `std::vector` or `std::string`. But there is no reason to keep `std::deque` in non-`constexpr` state since one of the main directions of C++ evolution is compile-time programming. And we want to use in compile-time as much as possible from STL. And this paper makes `std::deque` available in compile-time.

4 Proposed wording

We basically mark all the member and non-member functions of `std::deque` `constexpr`.

Direction to the editor: please apply `constexpr` to all of `std::deque`, including any additions that might be missing from this paper.

In **[support.limits.general]**, add the new feature test macro `__cpp_lib_constexpr_deque` with the corresponding value for header `<deque>` to Table 36 **[tab:support.ft]**.

Change in **[deque.syn] 22.3.3**:

```
#include <initializer_list>

namespace std {
  // 22.3.10, class template deque
  template<class T, class Allocator = allocator<T>> class deque;

  template<class T, class Allocator>
    constexpr bool operator==(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
  template<class T, class Allocator>
    constexpr synth-three-way-result<T> operator<=>(const deque<T, Allocator>& x, const deque<T,

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

  template<class T, class Allocator, class U>
    constexpr void erase(deque<T, Allocator>& c, const U& value);
  template<class T, class Allocator, class Predicate>
    constexpr void erase_if(deque<T, Allocator>& c, Predicate pred);

  [...]
}
```

Add after [**deque.overview**] **22.3.8.1/2**:

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

Change in [**deque.overview**] **22.3.8.1**:

```
namespace std {
  template<class T, class Allocator = allocator<T>>
  class deque {
  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 22.2
    using difference_type        = implementation-defined; // see 22.2
    using iterator               = implementation-defined; // see 22.2
    using const_iterator         = implementation-defined; // see 22.2
    using reverse_iterator       = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;

    // 22.3.8.2, construct/copy/destroy
    constexpr deque() : deque(Allocator()) { }
    constexpr explicit deque(const Allocator&);
    constexpr explicit deque(size_type n, const Allocator& = Allocator());
```

```
constexpr deque(size_type n, const T& value, const Allocator& = Allocator());
template<class InputIterator>
  constexpr deque(InputIterator first, InputIterator last, const Allocator& = Allocator());
constexpr deque(const deque& x);
constexpr deque(deque&& x);
constexpr deque(const deque& x, const Allocator&);
constexpr deque(deque&& x, const Allocator&);
constexpr deque(initializer_list<T>, const Allocator& = Allocator());

constexpr ~deque();
constexpr deque& operator=(const deque& x);
constexpr deque& operator=(deque&& x)
  noexcept(allocator_traits<Allocator>::is_always_equal::value);
constexpr deque& 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;

// 22.3.8.3, capacity
[[nodiscard]] constexpr bool empty() const noexcept;
constexpr size_type size() const noexcept;
constexpr size_type max_size() const noexcept;
constexpr void      resize(size_type sz);
constexpr void      resize(size_type sz, const T& c);
constexpr void      shrink_to_fit();

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

```
    constexpr const_reference back() const;

    // 22.3.8.4, modifiers
    template<class... Args> constexpr reference emplace_front(Args&&... args);
    template<class... Args> constexpr reference emplace_back(Args&&... args);
    template<class... Args> constexpr iterator emplace(const_iterator position, Args&&... args);

    constexpr void push_front(const T& x);
    constexpr void push_front(T&& x);
    constexpr void push_back(const T& x);
    constexpr void push_back(T&& x);

    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>);

    constexpr void pop_front();
    constexpr void pop_back();

    constexpr iterator erase(const_iterator position);
    constexpr iterator erase(const_iterator first, const_iterator last);
    constexpr void     swap(deque&)
      noexcept(allocator_traits<Allocator>::is_always_equal::value);
    constexpr void     clear() noexcept;
  };

  template<class InputIterator,
           class Allocator = allocator<iter-value-type<InputIterator>>>
    deque(InputIterator, InputIterator, Allocator = Allocator())
      -> deque<iter-value-type<InputIterator>, Allocator>;

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

Change in **[deque.cons] 22.3.8.2**:

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

[...]

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

[...]

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

[...]

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

[...]

Change in [**deque.capacity**] **22.3.8.3**:

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

[...]

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

[...]

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

[...]

Change in [**deque.modifiers**] **22.3.8.4**:

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

[...]

Change in [**deque.erasure**] **22.3.8.5**:

```
template<class T, class Allocator, class U>
  constexpr void erase(deque<T, Allocator>& c, const U& value);

template<class T, class Allocator, class Predicate>
  constexpr void erase_if(deque<T, Allocator>& c, Predicate pred);
```

5 Implementation

Possible implementation can be found here: LLVM fork. Notice that when proposal was written constexpr destructors were not supported in Clang. Also in this implementation isn't used `operator<=>` - bunch of old operators used instead (just because libcxx at the moment doesn't use `operator<=>` for `std::deque`).

6 References

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