Making `std::list` constexpr

Document #:	P0000
Date:	2019-09-28
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::list` is not currently `constexpr` friendly. With the loosening of requirements on `constexpr` in [P0784R1] and related papers, we can now make `std::list` `constexpr`, and we should in order to support the `constexpr` reflection effort (and other evident use cases).

3 Motivation

`std::list` is not so widely-used standard container as `std::vector` or `std::string`. But there is no reason to keep `std::list` 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::list` available in compile-time.

4 Proposed wording

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

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

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

Change in [**list.syn**] **22.3.5**:

```
#include <initializer_list>

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

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

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

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

  [...]
}
```

Add after [**list.overview**] **22.3.10.1/2**:

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

Change in [**list.overview**] **22.3.10.1**:

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

    // 22.3.10.2, construct/copy/destroy
    constexpr list() : list(Allocator()) { }
    constexpr explicit list(const Allocator&);
    constexpr explicit list(size_type n, const Allocator& = Allocator());
```

```
constexpr list(size_type n, const T& value, const Allocator& = Allocator());
template<class InputIterator>
  constexpr list(InputIterator first, InputIterator last, const Allocator& = Allocator());
constexpr list(const list& x);
constexpr list(list&&);
constexpr list(const list&, const Allocator&);
constexpr list(list&&, const Allocator&);
constexpr list(initializer_list<T>, const Allocator& = Allocator());
constexpr ~list();
constexpr list& operator=(const list& x);
constexpr list& operator=(list&& x)
  noexcept(allocator_traits<Allocator>::is_always_equal::value);
constexpr list& 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;

// 22.3.10.3, 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);

// element access
constexpr reference       front();
constexpr const_reference front() const;
constexpr reference       back();
constexpr const_reference back() const;

// 22.3.10.4, modifiers
template<class... Args> constexpr reference emplace_front(Args&&... args);
template<class... Args> constexpr reference emplace_back(Args&&... args);
constexpr void push_front(const T& x);
```

```
  constexpr void push_front(T&& x);
  constexpr void pop_front();
  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(list&)
    noexcept(allocator_traits<Allocator>::is_always_equal::value);
  constexpr void     clear() noexcept;

  // 22.3.10.5, list operations
  constexpr void splice(const_iterator position, list& x);
  constexpr void splice(const_iterator position, list&& x);
  constexpr void splice(const_iterator position, list& x, const_iterator i);
  constexpr void splice(const_iterator position, list&& x, const_iterator i);
  constexpr void splice(const_iterator position, list& x, const_iterator first, const_iterator
  constexpr void splice(const_iterator position, list&& x, const_iterator first, const_iterato

  constexpr size_type remove(const T& value);
  template<class Predicate> constexpr size_type remove_if(Predicate pred);

  constexpr size_type unique();
  template<class BinaryPredicate>
  constexpr size_type unique(BinaryPredicate binary_pred);

  constexpr void merge(list& x);
  constexpr void merge(list&& x);
  template<class Compare> constexpr void merge(list& x, Compare comp);
  template<class Compare> constexpr void merge(list&& x, Compare comp);

  constexpr void sort();
  template<class Compare> constexpr void sort(Compare comp);

  constexpr void reverse() noexcept;
};

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

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

Change in [**list.cons**] **22.3.10.2**:

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

[...]

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

[...]

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

[...]

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

[...]

Change in [**list.capacity**] **22.3.10.3**:

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

[...]

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

[...]

Change in [**list.modifiers**] **22.3.10.4**:

```
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();
constexpr void clear() noexcept;
```

Change in [**list.ops**] **22.3.10.5**:

```
constexpr void splice(const_iterator position, list& x);
constexpr void splice(const_iterator position, list&& x);

constexpr void splice(const_iterator position, list& x, const_iterator i);
constexpr void splice(const_iterator position, list&& x, const_iterator i);

constexpr void splice(const_iterator position, list& x, const_iterator first,
const_iterator last);
constexpr void splice(const_iterator position, list&& x, const_iterator first,
const_iterator last);
```

[...]

```
constexpr size_type remove(const T& value);
template<class Predicate> constexpr size_type remove_if(Predicate pred);
```

[...]

```
constexpr size_type unique();
template<class BinaryPredicate> constexpr size_type unique(BinaryPredicate binary_pred);
```

[...]

```
constexpr void merge(list& x);
constexpr void merge(list&& x);
template<class Compare> constexpr void merge(list& x, Compare comp);
template<class Compare> constexpr void merge(list&& x, Compare comp);
```

[...]

```
constexpr void reverse() noexcept;
```

[...]

```
constexpr void sort();
template<class Compare> constexpr void sort(Compare comp);
```

Change in [**list.erasure**] **22.3.10.6**:

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

template<class T, class Allocator, class Predicate>
  constexpr void erase_if(list<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::forward_list`).

6 References

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