Making `std::priority_queue` constexpr

Document #:	P0000
Date:	2019-09-23
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::priority_queue` is not currently `constexpr` friendly. With the loosening of requirements on `constexpr` in [P0784R1] and related papers, we can now make `std::priority_queue` `constexpr`, and we should in order to support the `constexpr` reflection effort (and other evident use cases).

3 Motivation

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

4 Proposed wording

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

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

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

Change in **[queue.syn] 22.6.2**:

```
#include <initializer_list>

namespace std {
  [...]
  template<class T, class Container = vector<T>,
           class Compare = less<typename Container::value_type>>
    class priority_queue;

  template<class T, class Container, class Compare>
    constexpr void swap(priority_queue<T, Container, Compare>& x,
      priority_queue<T, Container, Compare>& y) noexcept(noexcept(x.swap(y)));
  template<class T, class Container, class Compare, class Alloc>
    struct uses_allocator<priority_queue<T, Container, Compare>, Alloc>;
}
```

[...]

Change in [**priqueue.overview**] **22.6.5.1**:

```
namespace std {
  template<class T, class Container = vector<T>,
           class Compare = less<typename Container::value_type>>
  class priority_queue {
  public:
        using value_type      = typename Container::value_type;
        using reference       = typename Container::reference;
        using const_reference = typename Container::const_reference;
        using size_type       = typename Container::size_type;
        using container_type  = Container;
        using value_compare   = Compare;

  protected:
        Container c;
        Compare comp;

  public:
        constexpr priority_queue() : priority_queue(Compare()) {}
        constexpr explicit priority_queue(const Compare& x) : priority_queue(x, Container()) {}
        constexpr priority_queue(const Compare& x, const Container&);
        constexpr priority_queue(const Compare& x, Container&&);
        template<class InputIterator>
          constexpr priority_queue(InputIterator first, InputIterator last, const Compare& x,
        const Container&);
        template<class InputIterator>
          constexpr priority_queue(InputIterator first, InputIterator last,
        const Compare& x = Compare(), Container&& = Container());
        template<class Alloc> constexpr explicit priority_queue(const Alloc&);
        template<class Alloc> constexpr priority_queue(const Compare&, const Alloc&);
        template<class Alloc> constexpr priority_queue(const Compare&, const Container&, const A
        template<class Alloc> constexpr priority_queue(const Compare&, Container&&, const Alloc&
        template<class Alloc> constexpr priority_queue(const priority_queue&, const Alloc&);
        template<class Alloc> constexpr priority_queue(priority_queue&&, const Alloc&);
```

```
      [[nodiscard]] constexpr bool empty() const { return c.empty(); }
      constexpr size_type size()  const          { return c.size(); }
      constexpr const_reference   top() const    { return c.front(); }
      constexpr void push(const value_type& x);
      constexpr void push(value_type&& x);
      template<class... Args> constexpr void emplace(Args&&... args);
      constexpr void pop();
      constexpr void swap(priority_queue& q) noexcept(is_nothrow_swappable_v<Container> &&
        is_nothrow_swappable_v<Compare>)
          { using std::swap; swap(c, q.c); swap(comp, q.comp); }
  };

  template<class Compare, class Container>
    priority_queue(Compare, Container)
      -> priority_queue<typename Container::value_type, Container, Compare>;

  template<class InputIterator,
           class Compare = less<typename iterator_traits<InputIterator>::value_type>,
           class Container = vector<typename iterator_traits<InputIterator>::value_type>>
    priority_queue(InputIterator, InputIterator, Compare = Compare(), Container = Container())
      -> priority_queue<typename iterator_traits<InputIterator>::value_type, Container, Compare>

  template<class Compare, class Container, class Allocator>
    priority_queue(Compare, Container, Allocator)
      -> priority_queue<typename Container::value_type, Container, Compare>;

  // no equality is provided

  template<class T, class Container, class Compare>
    constexpr void swap(priority_queue<T, Container, Compare>& x,
      priority_queue<T, Container, Compare>& y) noexcept(noexcept(x.swap(y)));

  template<class T, class Container, class Compare, class Alloc>
    struct uses_allocator<priority_queue<T, Container, Compare>, Alloc>
      : uses_allocator<Container, Alloc>::type { };
}
```

Change in [**priqueue.cons**] **22.6.5.2**:

```
constexpr priority_queue(const Compare& x, const Container& y);
constexpr priority_queue(const Compare& x, Container&& y);
```

[...]

```
template<class InputIterator>
  constexpr priority_queue(InputIterator first, InputIterator last,
                           const Compare& x, const Container& y);
template<class InputIterator>
  constexpr priority_queue(InputIterator first, InputIterator last,
                           const Compare& x = Compare(), Container&& y = Container());
```

Change in [**priqueue.cons.alloc**] **22.6.5.3**:

```
template<class Alloc> constexpr explicit priority_queue(const Alloc& a);
```

[...]

```
template<class Alloc> constexpr priority_queue(const Compare& compare, const Alloc& a);
```

[...]

```
template<class Alloc>
  constexpr priority_queue(const Compare& compare, const Container& cont, const Alloc& a);
```

[...]

```
template<class Alloc>
  constexpr priority_queue(const Compare& compare, Container&& cont, const Alloc& a);
```

[...]

```
template<class Alloc> constexpr priority_queue(const priority_queue& q, const Alloc& a);
```

[...]

```
template<class Alloc> constexpr priority_queue(priority_queue&& q, const Alloc& a);
```

[...]

Change in [**priqueue.members**] **22.6.5.4**:

```
constexpr void push(const value_type& x);
```

[...]

```
constexpr void push(value_type&& x);
```

[...]

```
template<class... Args> constexpr void emplace(Args&&... args)
```

[...]

```
constexpr void pop();
```

[...]

Change in [**priqueue.special**] **22.6.5.5**:

```
template<class T, class Container, class Compare>
  constexpr void swap(priority_queue<T, Container, Compare>& x,
    priority_queue<T, Container, Compare>& y) noexcept(noexcept(x.swap(y)));
```

5 Implementation

Possible implementation can be found here: LLVM fork. Notice that when proposal was written constexpr destructors were not supported in Clang.

6 References

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