Making `std::stack` constexpr

1 Revision history

- R0 – Initial draft

2 Abstract

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

3 Motivation

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

4 Proposed wording

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

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

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

Change in [**stack.syn**] **22.6.3**:

```
#include <initializer_list>

namespace std {
  template<class T, class Container = deque<T>> class stack;

  template<class T, class Container>
    constexpr bool operator==(const stack<T, Container>& x, const stack<T, Container>& y);
  template<class T, class Container>
    constexpr bool operator!=(const stack<T, Container>& x, const stack<T, Container>& y);
  template<class T, class Container>
    constexpr bool operator< (const stack<T, Container>& x, const stack<T, Container>& y);
  template<class T, class Container>
    constexpr bool operator> (const stack<T, Container>& x, const stack<T, Container>& y);
  template<class T, class Container>
    constexpr bool operator<=(const stack<T, Container>& x, const stack<T, Container>& y);
  template<class T, class Container>
    constexpr bool operator>=(const stack<T, Container>& x, const stack<T, Container>& y);
  template<class T, three_way_comparable Container>
    constexpr compare_three_way_result_t<Container>
      operator<=>(const stack<T, Container>& x, const stack<T, Container>& y);

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

Change in [**stack.defn**] **22.6.6.1**:

```
namespace std {
  template<class T, class Container = deque<T>>
  class stack {
  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;

  protected:
        Container c;

  public:
        constexpr stack() : stack(Container()) {}
        constexpr explicit stack(const Container&);
        constexpr explicit stack(Container&&);
        template<class Alloc> constexpr explicit stack(const Alloc&);
        template<class Alloc> constexpr stack(const Container&, const Alloc&);
        template<class Alloc> constexpr stack(Container&&, const Alloc&);
        template<class Alloc> constexpr stack(const stack&, const Alloc&);
        template<class Alloc> constexpr stack(stack&&, const Alloc&);
```

```
                [[nodiscard]] constexpr bool empty() const      { return c.empty(); }
                constexpr size_type size()   const              { return c.size(); }
                constexpr reference          top()              { return c.back(); }
                constexpr const_reference    top() const        { return c.back(); }
                constexpr void push(const value_type& x)        { c.push_back(x); }
                constexpr void push(value_type&& x)             { c.push_back(std::move(x)); }
                template<class... Args>
                  constexpr decltype(auto) emplace(Args&&... args)
                    { return c.emplace_back(std::forward<Args>(args)...); }
                constexpr void pop()                            { c.pop_back(); }
                constexpr void swap(stack& s) noexcept(is_nothrow_swappable_v<Container>)
                  { using std::swap; swap(c, s.c); }
            };

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

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

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

Change in [**stack.cons**] **22.6.6.2**:

```
    constexpr explicit stack(const Container& cont);
```

[...]

```
    constexpr explicit stack(Container&& cont);
```

Change in [**stack.cons.alloc**] **22.6.6.3**:

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

[...]

```
    template<class Alloc> constexpr stack(const container_type& cont, const Alloc& a);
```

[...]

```
    template<class Alloc> constexpr stack(container_type&& cont, const Alloc& a);
```

[...]

```
    template<class Alloc> constexpr stack(const stack& s, const Alloc& a);
```

[...]

```
    template<class Alloc> constexpr stack(stack&& s, const Alloc& a);
```

[...]

Change in [**stack.ops**] **22.6.6.4**:

```
template<class T, class Container>
  constexpr bool operator==(const stack<T, Container>& x, const stack<T, Container>& y);
```

[...]

```
template<class T, class Container>
  constexpr bool operator!=(const stack<T, Container>& x, const stack<T, Container>& y);
```

[...]

```
template<class T, class Container>
  constexpr bool operator< (const stack<T, Container>& x, const stack<T, Container>& y);
```

[...]

```
template<class T, class Container>
  constexpr bool operator> (const stack<T, Container>& x, const stack<T, Container>& y);
```

[...]

```
template<class T, class Container>
  constexpr bool operator<=(const stack<T, Container>& x, const stack<T, Container>& y);
```

[...]

```
template<class T, class Container>
  constexpr bool operator>=(const stack<T, Container>& x, const stack<T, Container>& y);
```

[...]

```
template<class T, three_way_comparable Container>
  constexpr compare_three_way_result_t<Container>
    operator<=>(const stack<T, Container>& x, const stack<T, Container>& y);
```

[...]

Change in [**stack.special**] **22.6.6.5**:

```
template<class T, class Container>
  constexpr void swap(stack<T, Container>& x, stack<T, Container>& 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