

Iterators pair constructors for stack and queue

Document #: P1425R3
Date: 2021-02-18
Project: Programming Language C++
Audience: LEWG, LWG
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract

This paper proposes to add iterators-pair constructors to `std::stack` and `std::queue`

Tony tables

Before	After
<pre>std::vector<int> v(42); std::queue<int> q({v.begin(), v.end()}); std::stack<int> s({v.begin(), v.end()});</pre>	<pre>std::vector<int> v(42); std::queue q(v.begin(), v.end()); std::stack s(v.begin(), v.end());</pre>

Revisions

R3

- Add missing deduction guides for the constructors with allocators.
Since LEWG review, the deduction guides have been fixed to deduce the allocator type of the default container.

R2

- Remove the Container argument
- Add allocator support - in alignment with [LWG3506](#) [?]
- Add feature test macros

Motivation

`std::stack` and `std::queue` do not provide iterators based constructors which is inconsistent. This paper is an offshoot of [\[P1206\]](#), for which I conducted a review of existing containers and containers adapters constructors.

The lack of these constructors forces the implementation of `ranges::to` to special case container-adapters or to not support them. Their absence make it also impossible to deduce their type using CTAD.

While this is a a small change, we believe its impact on the standard is low and consistent designs are less surprising and therefore easier to use: with this change, all container-like types, whether they are *Containers* or container adapters, can be constructed from an iterators pair, making them more compatible with `ranges`.

Removal of the Container argument in R2

Previous iteration had a `queue(Iterator first, Iterator last, Container c)` argument, which was added for consistency with `priority_queue`. However, it was specified to insert the range denoted by `[first, last)` at the end of `c`. LWG astutely noted that this was confusing and wanted LEWG's input.

As a result, i decided to remove this argument entirely, as I can't think of a non-confusing fix, nor can I really come up with a good justification for this parameter.

- Changing the order of parameters would be inconsistent with `priority_queue`.
- Mandating that the range is inserted at the beginning of the container has performance drawbacks.
- It is unclear that using the order of parameter to determine the order of insertion would actually make sense to users.

Implementation

This proposal has been [Implemented in libc++](#)

Proposed Wording



Definition

[queue.defn]

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

```

protected:
Container c;

public:
queue() : queue(Container()) {}
explicit queue(const Container&);
explicit queue(Container&&);

template<class InputIterator>
queue(InputIterator first, InputIterator last);

template<class InputIterator, class Alloc>
queue(InputIterator first, InputIterator last, const Alloc&);

template<class Alloc> explicit queue(const Alloc&);
template<class Alloc> queue(const Container&, const Alloc&);
template<class Alloc> queue(Container&&, const Alloc&);
template<class Alloc> queue(const queue&, const Alloc&);
template<class Alloc> queue(queue&&, const Alloc&);

//...
};

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

template<class InputIterator>
queue(InputIterator, InputIterator)
-> queue<typename iterator_traits<InputIterator>::value_type>;

template<class InputIterator, class Allocator>
queue(InputIterator, InputIterator, Allocator)
-> queue<typename iterator_traits<InputIterator>::value_type,
deque<typename iterator_traits<InputIterator>::value_type, Allocator>>;

```

Note for LWG: Align the deduction guide with the outcome of P1518R0 [?] ?

```

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

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

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

```



Constructors

[queue.cons]

```
explicit queue(const Container& cont);
```

Effects: Initializes `c` with `cont`.

```
explicit queue(Container&& cont);
```

Effects: Initializes `c` with `std::move(cont)`.

```
template<class InputIterator>  
queue(InputIterator first, InputIterator last);
```

Effects: Initializes `c` with `first` as the first argument, and `last` as the second argument.

```
template<class InputIterator, typename Alloc>  
queue(InputIterator first, InputIterator last, const Alloc & alloc);
```

Effects: Initializes `c` with `first` as the first argument, `last` as the second argument and `alloc` as the third argument.



Definition

[stack.defn]

```
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:  
        stack() : stack(Container()) {}  
        explicit stack(const Container&);  
        explicit stack(Container&&);  
  
        template<class InputIterator>  
        stack(InputIterator first, InputIterator last);  
  
        template<class InputIterator, class Alloc>  
        stack(InputIterator first, InputIterator last, const Alloc&);  
  
        template<class Alloc> explicit stack(const Alloc&);  
        template<class Alloc> stack(const Container&, const Alloc&);  
        template<class Alloc> stack(Container&&, const Alloc&);  
        template<class Alloc> stack(const stack&, const Alloc&);  
        template<class Alloc> stack(stack&&, const Alloc&);  
};
```

```

        //...
};

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

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

template<class InputIterator, class Allocator>
stack(InputIterator, InputIterator, Allocator)
-> stack<typename iterator_traits<InputIterator>::value_type,
    deque<typename iterator_traits<InputIterator>::value_type, Allocator>>;

```

Note for LWG: Align the deduction guide with the outcome of P1518R0 [?] ?

```

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 { };
}

```



Constructors

[stack.cons]

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

Effects: Initializes c with cont.

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

Effects: Initializes c with `std::move(cont)`.

```
template<class InputIterator>
stack(InputIterator first, InputIterator last);
```

Effects: Initializes c with first as the first argument and last as the second argument.

```
template<class InputIterator, typename Alloc>
stack(InputIterator first, InputIterator last, const Alloc & alloc);
```

Effects: Initializes c with first as the first argument, last as the second argument and alloc as the third argument.

Feature test macros

Insert into [version.syn]

```
#define __cpp_lib_queue_iterators_constructor <DATE OF ADOPTION>  
#define __cpp_lib_stack_iterators_constructor <DATE OF ADOPTION>
```

Acknowledgment

Thanks to Eric Niebler who reviewed the wording

References

[P1206] Corentin Jabot *A function to convert any range to a container*
<https://wg21.link/P1206>