Iterators pair constructors for stack and queue

Document #:	P1425R4
Date:	2021-03-05
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

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

Revisions

R4

- Replace the 2 previously suggested feature macro by `__cpp_lib_adaptor_iterator_pair_constructor`
- Reorganize the ordfer of constructructors and put the allocators costructors on the approriate section
- use *`iter-value-type`* as introduced by LWG3506 [**?**]

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 entierly, 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 begining 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

Note for LWG & the editors: this wording uses the exposition only *iter-value-type* type which is introduced by the accepted resolution to issue LWG3506 [?]

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

                //...
        };

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


        template<class InputIterator>
        queue(InputIterator, InputIterator) -> queue<iter-value-type<InputIterator>>;

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

```
        template<class InputIterator, class Allocator>
        queue(InputIterator, InputIterator, Allocator)
        -> queue<iter-value-type<InputIterator>, deque<iter-value-type<InputIterator>, Allocator>>;


        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.

🔗 Constructors with allocators [queue.cons.alloc]

[...]

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

Effects: Initializes `c` with `q.c` as the first argument and `a` as the second argument.

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

Effects: Initializes `c` with `std::move(q.c)` as the first argument and `a` as the second argument.

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

                //...
        };

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

        template<class InputIterator>
        stack(InputIterator, InputIterator)
        -> stack<iter-value-type<InputIterator>>;

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

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


        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.

🔲 Constructors with allocators [stack.cons.alloc]

[...]

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

Effects: Initializes `c` with `s.c` as the first argument and `a` as the second argument.

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

Effects: Initializes `c` with `std::move(s.c)` as the first argument and `a` as the second argument.

```
template<class InputIterator, class 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_adaptor_iterator_pair_constructor <DATE OF ADOPTION> // also in <stack> and <queue>
```

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