std::deque

Defined in header <deque>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class deque;
```

`std::deque` (double-ended queue) is an indexed sequence container that allows fast insertion and deletion at both its beginning and its end. In addition, insertion and deletion at either end of a deque never invalidates pointers or references to the rest of the elements.

As opposed to `std::vector`, the elements of a deque are not stored contiguously: typical implementations use a sequence of individually allocated fixed-size arrays.

The storage of a deque is automatically expanded and contracted as needed. Expansion of a deque is cheaper than the expansion of a `std::vector` because it does not involve copying of the existing elements to a new memory location.

The complexity (efficiency) of common operations on deques is as follows:

- Random access - constant *O(1)*
- Insertion or removal of elements at the end or beginning - constant *O(1)*
- Insertion or removal of elements - linear *O(n)*

`std::deque` meets the requirements of `Container`, `AllocatorAwareContainer`, `SequenceContainer` and `ReversibleContainer`.

Template parameters

`T` - The type of the elements.

`T` must meet the requirements of `CopyAssignable` and `CopyConstructible`.	(until C++11)
The requirements that are imposed on the elements depend on the actual operations performed on the container. Generally, it is required that element type is a complete type and meets the requirements of `Erasable`, but many member functions impose stricter requirements.	(since C++11)

`Allocator` - An allocator that is used to acquire memory to store the elements. The type must meet the requirements of `Allocator`.

Iterator invalidation

This section is incomplete

There are still a few inaccuracies in this section, refer to individual member function pages for more detail

Operations	Invalidated
All read only operations, `swap`, `std::swap`	Never
`shrink_to_fit`, `clear`, `insert`, `emplace`, `push_back`, `emplace_back`	Always
`erase`	If erasing at beginning or end - only erased elements. Otherwise - all iterators are invalidated.
`resize`	Only if the new size is bigger than the old one.
`pop_back`, `pop_front`	Only to the element erased

Notes

- Under some circumstances, references are not invalidated by `insert` and `emplace`.
- `push_back` and `emplace_back` do not invalidate any references.

Member types

Member type	Definition

`value_type`	`T`
`allocator_type`	`Allocator`
`size_type`	Unsigned integral type (usually `std::size_t`)
`difference_type`	Signed integer type (usually `std::ptrdiff_t`)
`reference`	`Allocator::reference` (until C++11) `value_type&` (since C++11)
`const_reference`	`Allocator::const_reference` (until C++11) `const value_type&` (since C++11)
`pointer`	`Allocator::pointer` (until C++11) `std::allocator_traits<Allocator>::pointer` (since C++11)
`const_pointer`	`Allocator::const_pointer` (until C++11) `std::allocator_traits<Allocator>::const_pointer` (since C++11)
`iterator`	`RandomAccessIterator`
`const_iterator`	Constant random access iterator
`reverse_iterator`	`std::reverse_iterator<iterator>`
`const_reverse_iterator`	`std::reverse_iterator<const_iterator>`

Member functions

(constructor)	constructs the `deque` (public member function)
(destructor)	destructs the `deque` (public member function)
`operator=`	assigns values to the container (public member function)
`assign`	assigns values to the container (public member function)
`get_allocator`	returns the associated allocator (public member function)

Element access

`at`	access specified element with bounds checking (public member function)
`operator[]`	access specified element (public member function)
`front`	access the first element (public member function)
`back`	access the last element (public member function)

Iterators

`begin` **`cbegin`**	returns an iterator to the beginning (public member function)
`end` **`cend`**	returns an iterator to the end (public member function)
`rbegin` **`crbegin`**	returns a reverse iterator to the beginning (public member function)
`rend` **`crend`**	returns a reverse iterator to the end (public member function)

Capacity

`empty`	checks whether the container is empty (public member function)
`size`	returns the number of elements (public member function)
`max_size`	returns the maximum possible number of elements (public member function)
`shrink_to_fit` (C++11)	reduces memory usage by freeing unused memory (public member function)

Modifiers

clear	clears the contents (public member function)
insert	inserts elements (public member function)
emplace (C++11)	constructs element in-place (public member function)
erase	erases elements (public member function)
push_back	adds elements to the end (public member function)
emplace_back (C++11)	constructs elements in-place at the end (public member function)
pop_back	removes the last element (public member function)
push_front	inserts elements to the beginning (public member function)
emplace_front (C++11)	constructs elements in-place at the beginning (public member function)
pop_front	removes the first element (public member function)
resize	changes the number of elements stored (public member function)
swap	swaps the contents (public member function)

Non-member functions

operator== **operator!=** **operator<** **operator<=** **operator>** **operator>=**	lexicographically compares the values in the deque (function template)
std::swap(std::deque)	specializes the `std::swap` algorithm (function template)

std::deque::deque

`explicit deque( const Allocator& alloc = Allocator() );`	(1)	(until C++14)
`deque() : deque( Allocator() ) {}` `explicit deque( const Allocator& alloc );`	(1)	(since C++14)

`explicit deque( size_type count,`
`const T& value = T(),`
`const Allocator& alloc = Allocator());`
	(2)	(until C++11)

`deque( size_type count,`
`const T& value,`
`const Allocator& alloc = Allocator());`
	(2)	(since C++11)

`explicit deque( size_type count );`	(3)	(since C++11) (until C++14)
`explicit deque( size_type count, const Allocator& alloc = Allocator() );`	(3)	(since C++14)

`template< class InputIt >`
`deque( InputIt first, InputIt last,`
`const Allocator& alloc = Allocator() );`
	(4)	

`deque( const deque& other );`	(5)	
`deque( const deque& other, const Allocator& alloc );`	(5)	(since C++11)
`deque( deque&& other )`	(6)	(since C++11)
`deque( deque&& other, const Allocator& alloc );`	(6)	(since C++11)
`deque( std::initializer_list<T> init,` `const Allocator& alloc = Allocator() );`	(7)	(since C++11)

Constructs a new container from a variety of data sources, optionally using a user supplied allocator `alloc`.

1) Default constructor. Constructs an empty container.
2) Constructs the container with `count` copies of elements with value `value`.
3) Constructs the container with `count` default-inserted instances of `T`. No copies are made.
4) Constructs the container with the contents of the range `[first, last)`.

This constructor has the same effect as overload (2) if `InputIt` is an integral type.	(until C++11)
This overload only participates in overload resolution if `InputIt` satisfies `InputIterator` , to avoid ambiguity with the overload (2).	(since C++11)

5) Copy constructor. Constructs the container with the copy of the contents of `other`. If `alloc` is not provided, allocator is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator()`.
6) Move constructor. Constructs the container with the contents of `other` using move semantics. If `alloc` is not provided, allocator is obtained by move-construction from the allocator belonging to `other`.
7) Constructs the container with the contents of the initializer list `init`.

Parameters

alloc	-	allocator to use for all memory allocations of this container
count	-	the size of the container
value	-	the value to initialize elements of the container with
first, last	-	the range to copy the elements from
other	-	another container to be used as source to initialize the elements of the container with
init	-	initializer list to initialize the elements of the container with

Complexity

1) Constant
2-3) Linear in `count`
4) Linear in distance between `first` and `last`
5) Linear in size of `other`
6) Constant. If `alloc` is given and `alloc != other.get_allocator()`, then linear.
7) Linear in size of `init`

Example

Run this code

```
#include <deque>
#include <string>
#include <iostream>

template<typename T>
std::ostream& operator<<(std::ostream& s, const std::deque<T>& v) {
    s.put('[');
    char comma[3] = {'\0', ' ', '\0'};
    for (const auto& e : v) {
        s << comma << e;
        comma[0] = ',';
    }
    return s << ']';
}

int main()
{
    // c++11 initializer list syntax:
    std::deque<std::string> words1 {"the", "frogurt", "is", "also", "cursed"};
    std::cout << "words1: " << words1 << '\n';

    // words2 == words1
    std::deque<std::string> words2(words1.begin(), words1.end());
    std::cout << "words2: " << words2 << '\n';

    // words3 == words1
    std::deque<std::string> words3(words1);
    std::cout << "words3: " << words3 << '\n';

    // words4 is {"Mo", "Mo", "Mo", "Mo", "Mo"}
    std::deque<std::string> words4(5, "Mo");
    std::cout << "words4: " << words4 << '\n';
}
```

Output:

```
words1: [the, frogurt, is, also, cursed]
words2: [the, frogurt, is, also, cursed]
words3: [the, frogurt, is, also, cursed]
words4: [Mo, Mo, Mo, Mo, Mo]
```

See also

assign	assigns values to the container (public member function)
operator=	assigns values to the container (public member function)

std::deque::~deque

```
~deque();
```

Destructs the container. The destructors of the elements are called and the used storage is deallocated. Note, that if the elements are pointers, the pointed-to objects are not destroyed.

Complexity

Linear in the size of the container.

std::deque::operator=

```
deque& operator=( const deque& other );                    (1)
deque& operator=( deque&& other );                         (2)  (since C++11)
deque& operator=( std::initializer_list<T> ilist );        (3)  (since C++11)
```

Replaces the contents of the container.

1) Copy assignment operator. Replaces the contents with a copy of the contents of `other`. If `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment()` is `true`, the target allocator is replaced by a copy of the source allocator. If the target and the source allocators do not compare equal, the target (`*this`) allocator is used to deallocate the memory, then `other`'s allocator is used to allocate it before copying the elements. (since C++11)
2) Move assignment operator. Replaces the contents with those of `other` using move semantics (i.e. the data in `other` is moved from `other` into this container). `other` is in a valid but unspecified state afterwards. If `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment()` is `true`, the target allocator is replaced by a copy of the source allocator. If it is `false` and the source and the target allocators do not compare equal, the target cannot take ownership of the source memory and must move-assign each element individually, allocating additional memory using its own allocator as needed.
3) Replaces the contents with those identified by initializer list `ilist`.

Parameters

other - another container to use as data source
ilist - initializer list to use as data source

Return value

`*this`

Complexity

1) Linear in the size of the `other`.
2) Constant unless `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment()` is false and the allocators do not compare equal (in which case linear).
3) Linear in the size of `ilist`.

Exceptions

(since C++17)

2) `noexcept` specification:

```
noexcept(std::allocator_traits<Allocator>::is_always_equal::value)
```

Example

The following code uses to assign one `std::deque` to another:

Run this code

```
#include <deque>
#include <iostream>

void display_sizes(const std::deque<int>& nums1,
                   const std::deque<int>& nums2,
```

```
                  const std::deque<int>& nums3)
{
    std::cout << "nums1: " << nums1.size()
              << " nums2: " << nums2.size()
              << " nums3: " << nums3.size() << '\n';
}

int main()
{
    std::deque<int> nums1 {3, 1, 4, 6, 5, 9};
    std::deque<int> nums2;
    std::deque<int> nums3;

    std::cout << "Initially:\n";
    display_sizes(nums1, nums2, nums3);

    // copy assignment copies data from nums1 to nums2
    nums2 = nums1;

    std::cout << "After assigment:\n";
    display_sizes(nums1, nums2, nums3);

    // move assignment moves data from nums1 to nums3,
    // modifying both nums1 and nums3
    nums3 = std::move(nums1);

    std::cout << "After move assigment:\n";
    display_sizes(nums1, nums2, nums3);
}
```

Output:

```
Initially:
nums1: 6 nums2: 0 nums3: 0
After assigment:
nums1: 6 nums2: 6 nums3: 0
After move assigment:
nums1: 0 nums2: 6 nums3: 6
```

See also

(constructor)	constructs the deque (public member function)
assign	assigns values to the container (public member function)

std::deque::assign

`void assign( size_type count, const T& value );`	(1)	
`template< class InputIt >` `void assign( InputIt first, InputIt last );`	(2)	
`void assign( std::initializer_list<T> ilist );`	(3)	(since C++11)

Replaces the contents of the container.

1) Replaces the contents with `count` copies of value `value`
2) Replaces the contents with copies of those in the range `[first, last)`.

This overload has the same effect as overload (1) if `InputIt` is an integral type.	(until C++11)
This overload only participates in overload resolution if `InputIt` satisfies `InputIterator`.	(since C++11)

3) Replaces the contents with the elements from the initializer list `ilist`.

Parameters

`count`	-	the new size of the container
`value`	-	the value to initialize elements of the container with
`first, last`	-	the range to copy the elements from
`ilist`	-	initializer list to copy the values from

Complexity

1) Linear in `count`
2) Linear in distance between `first` and `last`
3) Linear in `ilist.size()`

Example

The following code uses `assign` to add several characters to a `std::deque<char>`:

Run this code

```
#include <deque>
#include <iostream>

int main()
{
    std::deque<char> characters;

    characters.assign(5, 'a');

    for (char c : characters) {
        std::cout << c << '\n';
    }

    return 0;
}
```

Output:

```
a
a
a
a
a
```

See also

(constructor)	constructs the `deque` (public member function)

std::deque::get_allocator

```
allocator_type get_allocator() const;
```

Returns the allocator associated with the container.

Parameters

(none)

Return value

The associated allocator.

Complexity

Constant.

std::deque::front

```
reference front();
const_reference front() const;
```

Returns a reference to the first element in the container.

Calling `front` on an empty container is undefined.

Parameters

(none)

Return value

reference to the first element

Complexity

Constant

Notes

For a container `c`, the expression `c.front()` is equivalent to `*c.begin()`.

Example

The following code uses `front` to display the first element of a `std::deque<char>`:

Run this code

```
#include <deque>
#include <iostream>

int main()
{
    std::deque<char> letters {'o', 'm', 'g', 'w', 't', 'f'};

    if (!letters.empty()) {
        std::cout << "The first character is: " << letters.front() << '\n';
    }
}
```

Output:

```
The first character is o
```

See also

back	access the last element (public member function)

std::deque::back

```
reference back();
const_reference back() const;
```

Returns reference to the last element in the container.

Calling `back` on an empty container is undefined.

Parameters

(none)

Return value

Reference to the last element.

Complexity

Constant.

Notes

For a container `c`, the expression `return c.back();` is equivalent to
`{ auto tmp = c.end(); --tmp; return *tmp; }`

Example

The following code uses `back` to display the last element of a `std::deque<char>`:

Run this code

```
#include <deque>
#include <iostream>

int main()
{
    std::deque<char> letters {'o', 'm', 'g', 'w', 't', 'f'};
    if (!letters.empty()) {
        std::cout << "The last character is: " << letters.back() << '\n';
    }
}
```

Output:

```
The last character is f
```

See also

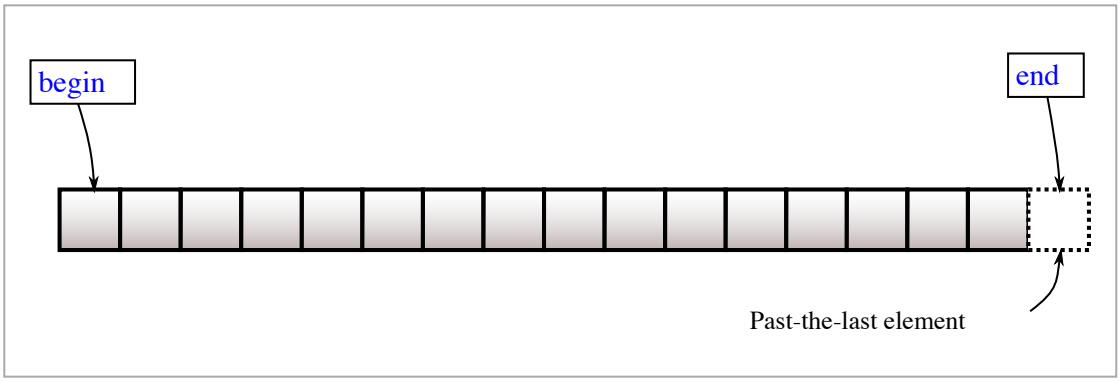
front	access the first element (public member function)

std::deque::begin, std::deque::cbegin

```
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;     (since C++11)
```

Returns an iterator to the first element of the container.

If the container is empty, the returned iterator will be equal to `end()`.



Parameters

(none)

Return value

Iterator to the first element

Exceptions

(none)		(until C++11)
`noexcept` specification:	`noexcept`	(since C++11)

Complexity

Constant

Example

This section is incomplete
Reason: no example

See also

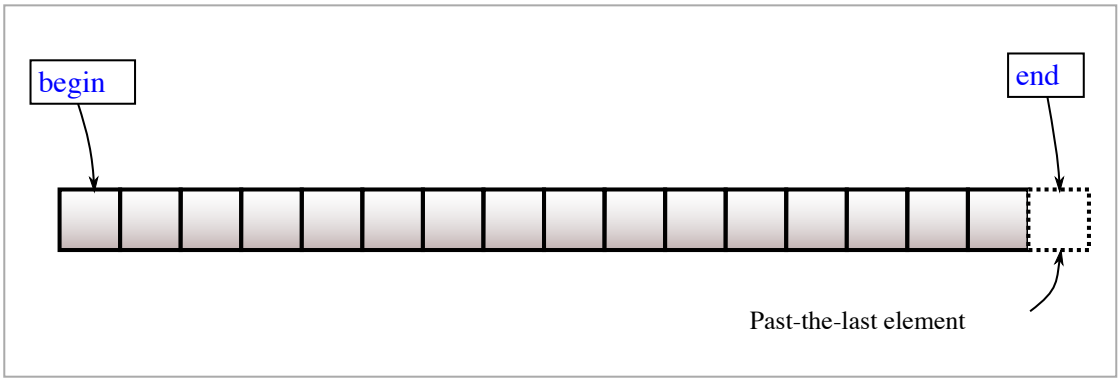
end	returns an iterator to the end
cend	(public member function)

std::deque::end, std::deque::cend

```
iterator end();
const_iterator end() const;
const_iterator cend() const;     (since C++11)
```

Returns an iterator to the element following the last element of the container.

This element acts as a placeholder; attempting to access it results in undefined behavior.



Parameters

(none)

Return value

Iterator to the element following the last element.

Exceptions

(none)		(until C++11)
`noexcept` specification:	`noexcept`	(since C++11)

Complexity

Constant.

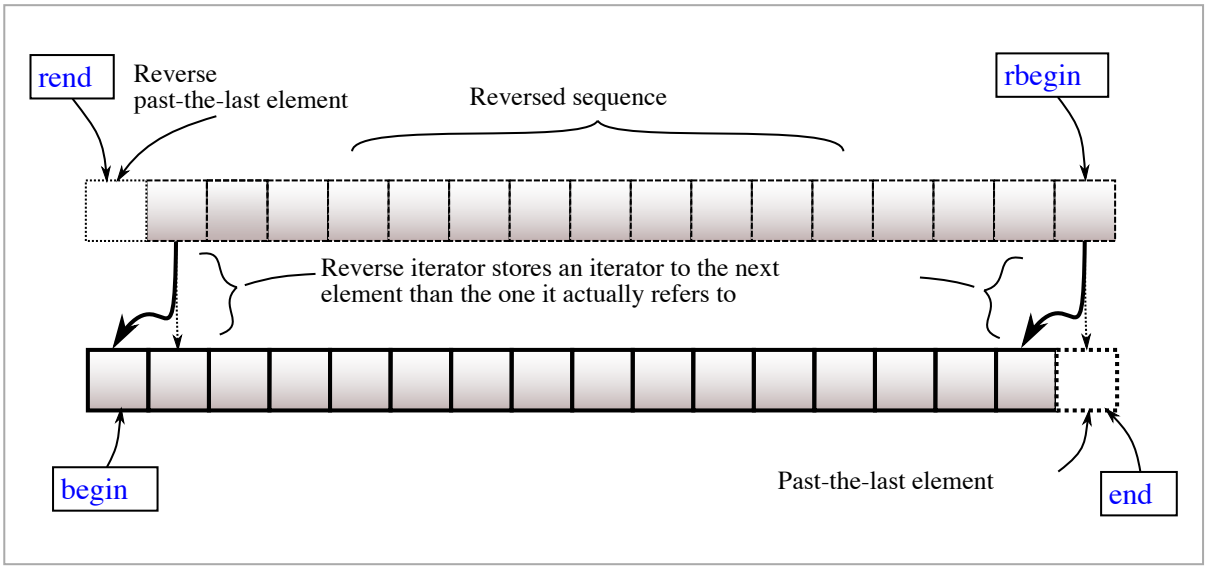
See also

`begin` **`cbegin`**	returns an iterator to the beginning (public member function)

std::deque::rbegin, std::deque::crbegin

```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
const_reverse_iterator crbegin() const;     (since C++11)
```

Returns a reverse iterator to the first element of the reversed container. It corresponds to the last element of the non-reversed container.



Parameters

(none)

Return value

Reverse iterator to the first element.

Exceptions

(none)		(until C++11)
`noexcept` specification:	`noexcept`	(since C++11)

Complexity

Constant.

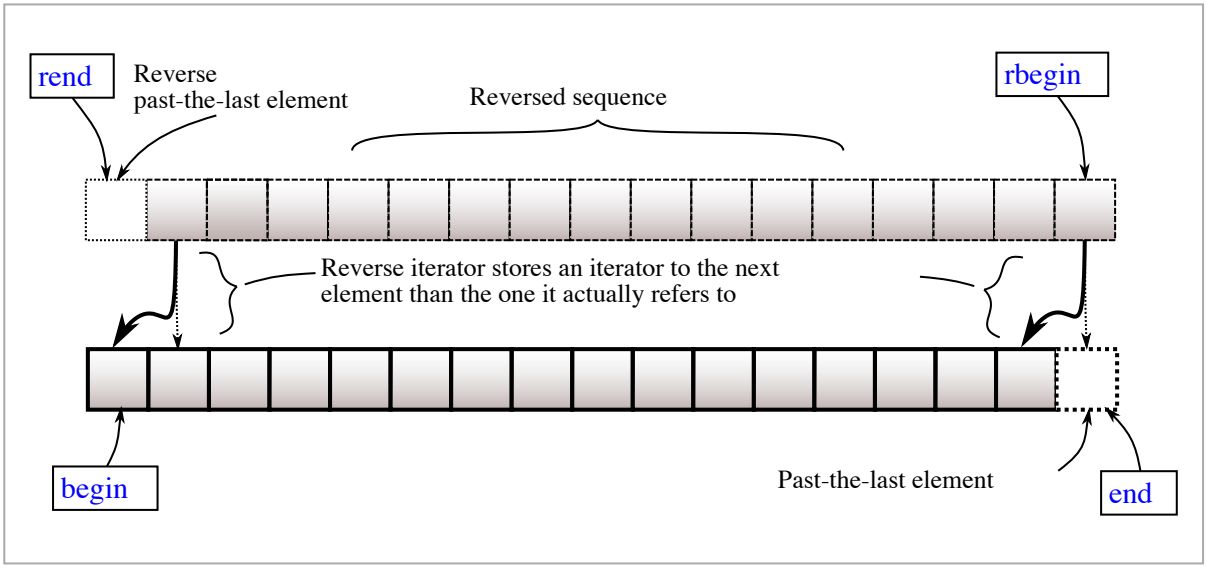
See also

rend **crend**	returns a reverse iterator to the end (public member function)

std::deque::rend, std::deque::crend

```
reverse_iterator rend();
const_reverse_iterator rend() const;
const_reverse_iterator crend() const;    (since C++11)
```

Returns a reverse iterator to the element following the last element of the reversed container. It corresponds to the element preceding the first element of the non-reversed container. This element acts as a placeholder, attempting to access it results in undefined behavior.



Parameters

(none)

Return value

Reverse iterator to the element following the last element.

Exceptions

(none)		(until C++11)
`noexcept` specification:	`noexcept`	(since C++11)

Complexity

Constant.

See also

`rbegin` **`crbegin`**	returns a reverse iterator to the beginning (public member function)

std::deque::empty

```
bool empty() const;
```

Checks if the container has no elements, i.e. whether `begin() == end()`.

Parameters

(none)

Return value

`true` if the container is empty, `false` otherwise

Exceptions

(none)		(until C++11)
`noexcept` specification:	`noexcept`	(since C++11)

Complexity

Constant.

Example

The following code uses `empty` to check if a `std::deque<int>` contains any elements:

Run this code

```
#include <deque>
#include <iostream>

int main()
{
    std::deque<int> numbers;
    std::cout << "Initially, numbers.empty(): " << numbers.empty() << '\n';

    numbers.push_back(42);
    numbers.push_back(13317);
    std::cout << "After adding elements, numbers.empty(): " << numbers.empty() << '\n';
}
```

Output:

```
Initially, numbers.empty(): 1
After adding elements, numbers.empty(): 0
```

See also

`size`	returns the number of elements (public member function)

std::deque::size

```
size_type size() const;
```

Returns the number of elements in the container, i.e. `std::distance(begin(), end())`.

Parameters

(none)

Return value

The number of elements in the container.

Exceptions

(none)	(until C++11)
`noexcept` specification: `noexcept`	(since C++11)

Complexity

Constant.

Example

The following code uses `size` to display the number of elements in a `std::deque`:

Run this code

```
#include <deque>
#include <iostream>

int main()
{
    std::deque<int> nums {1, 3, 5, 7};

    std::cout << "nums contains " << nums.size() << " elements.\n";
}
```

Output:

```
nums contains 4 elements.
```

See also

empty	checks whether the container is empty (public member function)
max_size	returns the maximum possible number of elements (public member function)
resize	changes the number of elements stored (public member function)

std::deque::max_size

```
size_type max_size() const;
```

Returns the maximum number of elements the container is able to hold due to system or library implementation limitations, i.e. `std::distance(begin(), end())` for the largest container.

Parameters

(none)

Return value

Maximum number of elements.

Exceptions

(none)		(until C++11)
`noexcept` specification:	`noexcept`	(since C++11)

Complexity

Constant.

Notes

This value is typically equal to `std::numeric_limits<size_type>::max()`, and reflects the theoretical limit on the size of the container. At runtime, the size of the container may be limited to a value smaller than `max_size()` by the amount of RAM available.

Example

Run this code

```
#include <iostream>
#include <deque>

int main()
{
    std::deque<char> s;
    std::cout << "Maximum size of a 'deque' is " << s.max_size() << "\n";
}
```

Possible output:

```
Maximum size of a 'deque' is 18446744073709551615
```

See also

`size`	returns the number of elements (public member function)

std::deque::shrink_to_fit

```
void shrink_to_fit();    (since C++11)
```

Requests the removal of unused capacity.

It is a non-binding request to reduce `capacity` to `size()`. It depends on the implementation if the request is fulfilled.

All iterators and references are potentially invalidated. Past-the-end iterator is also potentially invalidated.

Parameters

(none)

Type requirements

- `T` must meet the requirements of `MoveInsertable`.

Return value

(none)

Complexity

At most linear in the size of the container.

Example

Run this code

```
#include <deque>

int main() {
    std::deque<int> nums(1000, 42);
    nums.push_front(1);
    nums.pop_front();

    nums.clear();

    // nums now contains no items, but it may still be holding allocated memory.
    // Calling shrink_to_fit will free any unused memory.
    nums.shrink_to_fit();
}
```

See also

`size`	returns the number of elements (public member function)

std::deque::clear

```
void clear();
```

Removes all elements from the container.

Invalidates any references, pointers, or iterators referring to contained elements. May invalidate any past-the-end iterators.

Parameters

(none)

Return value

(none)

Exceptions

(none)		(until C++11)
`noexcept` specification:	`noexcept`	(since C++11)

Complexity

Linear in the size of the container.

`clear` is defined in terms of `erase`, which has linear complexity.	(until C++11)
complexity of `clear` is omitted	(since C++11) (until C++14)
`clear` has linear complexity for sequence containers.	(since C++14)

See also

`erase`	erases elements (public member function)

std::deque::insert

`iterator insert( iterator pos, const T& value );`	(1)	(until C++11)
`iterator insert( const_iterator pos, const T& value );`		(since C++11)
`iterator insert( const_iterator pos, T&& value );`	(2)	(since C++11)
`void insert( iterator pos, size_type count, const T& value );`	(3)	(until C++11)
`iterator insert( const_iterator pos, size_type count, const T& value );`		(since C++11)
`template< class InputIt >` `void insert( iterator pos, InputIt first, InputIt last);`	(4)	(until C++11)
`template< class InputIt >` `iterator insert( const_iterator pos, InputIt first, InputIt last );`		(since C++11)
`iterator insert( const_iterator pos, std::initializer_list<T> ilist );`	(5)	(since C++11)

Inserts elements at the specified location in the container.

1-2) inserts `value` before `pos`

3) inserts `count` copies of the `value` before `pos`

4) inserts elements from range `[first, last)` before `pos`.

This overload has the same effect as overload (3) if `InputIt` is an integral type.	(until C++11)
This overload only participates in overload resolution if `InputIt` qualifies as `InputIterator`, to avoid ambiguity with the overload (3).	(since C++11)

The behavior is undefined if `first` and `last` are iterators into `*this`.

5) inserts elements from initializer list `ilist` before `pos`.

All iterators, including the past-the-end iterator, are invalidated. References are invalidated too, unless `pos == begin()` or `pos == end()`, in which case they are not invalidated.

Parameters

`pos`	-	iterator before which the content will be inserted. `pos` may be the `end()` iterator
`value`	-	element value to insert
`first, last`	-	the range of elements to insert, can't be iterators into container for which insert is called
`ilist`	-	initializer list to insert the values from

Type requirements

- `T` must meet the requirements of `CopyAssignable` and `CopyInsertable` in order to use overload (1).
- `T` must meet the requirements of `MoveAssignable` and `MoveInsertable` in order to use overload (2).
- `T` must meet the requirements of `CopyAssignable` and `CopyInsertable` in order to use overload (3).
- `T` must meet the requirements of `EmplaceConstructible` in order to use overload (4,5).
- `T` must meet the requirements of `Swappable`, `MoveAssignable`, `MoveConstructible` and `MoveInsertable` in order to use overload (4,5). (since C++17)

Return value

1-2) Iterator pointing to the inserted `value`

3) Iterator pointing to the first element inserted, or `pos` if `count==0`.

4) Iterator pointing to the first element inserted, or `pos` if `first==last`.

5) Iterator pointing to the first element inserted, or `pos` if `ilist` is empty.

Complexity

1-2) Constant plus linear in the lesser of the distances between `pos` and either of the ends of the container.

3) Linear in `count` plus linear in the lesser of the distances between `pos` and either of the ends of the container.

4) Linear in `std::distance(first, last)` plus linear in the lesser of the distances between `pos` and either of the ends of the container.

5) Linear in `ilist.size()` plus linear in the lesser of the distances between `pos` and either of the ends of the container.

Exceptions

If an exception is thrown when inserting a single element at either end, this function has no effect (strong exception guarantee).

See also

`emplace` (C++11)	constructs element in-place (public member function)
`push_front`	inserts elements to the beginning (public member function)
`push_back`	adds elements to the end (public member function)

std::deque::emplace

```
template< class... Args >
iterator emplace( const_iterator pos, Args&&... args );
```
(since C++11)

Inserts a new element into the container directly before `pos`. The element is constructed through `std::allocator_traits::construct`, which typically uses placement-new to construct the element in-place at a location provided by the container. The arguments `args...` are forwarded to the constructor as `std::forward<Args>(args)...`.

All iterators, including the past-the-end iterator, are invalidated. References are invalidated too, unless `pos == begin()` or `pos == end()`, in which case they are not invalidated.

Parameters

pos - iterator before which the new element will be constructed

args - arguments to forward to the constructor of the element

Type requirements

- `T (the container's element type)` must meet the requirements of `MoveAssignable`, `MoveInsertable` and `EmplaceConstructible`.

Return value

Iterator pointing to the emplaced element.

Complexity

Linear in the lesser of the distances between `pos` and either of the ends of the container.

Exceptions

If an exception is thrown (e.g. by the constructor), the container is left unmodified, as if this function was never called (strong exception guarantee).

See also

insert	inserts elements (public member function)

std::deque::erase

`iterator erase( iterator pos );`	(1)	(until C++11)
`iterator erase( const_iterator pos );`		(since C++11)
`iterator erase( iterator first, iterator last );`	(2)	(until C++11)
`iterator erase( const_iterator first, const_iterator last );`		(since C++11)

Removes specified elements from the container.

1) Removes the element at `pos`.
2) Removes the elements in the range `[first; last)`.

All iterators and references are invalidated, unless the erased elements are at the end or the beginning of the container, in which case only the iterators and references to the erased elements are invalidated.

It is unspecified when the past-the-end iterator is invalidated.	(until C++11)
The past-the-end iterator is also invalidated unless the erased elements are at the beginning of the container and the last element is not erased.	(since C++11)

The iterator `pos` must be valid and dereferenceable. Thus the `end()` iterator (which is valid, but is not dereferencable) cannot be used as a value for `pos`.

The iterator `first` does not need to be dereferenceable if `first==last`: erasing an empty range is a no-op.

Parameters

pos	-	iterator to the element to remove
first, last	-	range of elements to remove

Type requirements

- `T` must meet the requirements of `MoveAssignable`.

Return value

Iterator following the last removed element. If the iterator `pos` refers to the last element, the `end()` iterator is returned.

Exceptions

Does not throw unless an exception is thrown by the copy constructor, move constructor, assignment operator, or move assignment operator of `T`.

Complexity

1) Linear in the lesser of the distances between `pos` and either of the ends of the container.
2) Linear in the distance between `first` and `last`, plus linear in the lesser of the number of elements before the erased elements and the number of elements after the erased elements.

Example

Run this code

```
#include <deque>
#include <iostream>


int main( )
{
    std::deque<int> c{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    for (auto &i : c) {
```

```
        std::cout << i << " ";
    }
    std::cout << '\n';

    c.erase(c.begin());

    for (auto &i : c) {
        std::cout << i << " ";
    }
    std::cout << '\n';

    c.erase(c.begin()+2, c.begin()+5);

    for (auto &i : c) {
        std::cout << i << " ";
    }
    std::cout << '\n';
}
```

Output:

```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 6 7 8 9
```

See also

clear	clears the contents (public member function)

std::deque::push_front

```
void push_front( const T& value );
void push_front( T&& value );      (since C++11)
```

Prepends the given element `value` to the beginning of the container.

All iterators, including the past-the-end iterator, are invalidated. No references are invalidated.

Parameters

`value` - the value of the element to prepend

Return value

(none)

Complexity

Constant.

Exceptions

If an exception is thrown, this function has no effect (strong exception guarantee).

See also

`emplace_front` (C++11)	constructs elements in-place at the beginning (public member function)
`push_back`	adds elements to the end (public member function)
`pop_front`	removes the first element (public member function)

std::deque::emplace_front

```
template< class... Args >
void emplace_front( Args&&... args );
```
(since C++11)

Inserts a new element to the beginning of the container. The element is constructed through `std::allocator_traits::construct`, which typically uses placement-new to construct the element in-place at the location provided by the container. The arguments `args...` are forwarded to the constructor as `std::forward<Args>(args)...`.

Parameters

`args` - arguments to forward to the constructor of the element

Type requirements

- `T (the container's element type)` must meet the requirements of `EmplaceConstructible`.

Return value

(none)

Complexity

Constant.

Exceptions

If an exception is thrown, this function has no effect (strong exception guarantee).

See also

`push_front`	inserts elements to the beginning (public member function)

std::deque::pop_front

```
void pop_front();
```

Removes the first element of the container.

Iterators and references to the erased element are invalidated. It is unspecified whether the past-the-end iterator is invalidated if the element is the last element in the container. Other references and iterators are not affected.	(since C++11)
Iterators and references to the erased element are invalidated. If the element is the last element in the container, the past-the-end iterator is also invalidated. Other references and iterators are not affected.	(until C++11)

Parameters

(none)

Return value

(none)

Complexity

Constant.

Exceptions

Does not throw.

See also

pop_back	removes the last element (public member function)
push_front	inserts elements to the beginning (public member function)

std::deque::push_back

```
void push_back( const T& value );     (1)
void push_back( T&& value );          (2)  (since C++11)
```

Appends the given element `value` to the end of the container.

1) The new element is initialized as a copy of `value`.
2) `value` is moved into the new element.

All iterators, including the past-the-end iterator, are invalidated. No references are invalidated.

Parameters

`value` - the value of the element to append

Type requirements

- `T` must meet the requirements of `CopyInsertable` in order to use overload (1).
- `T` must meet the requirements of `MoveInsertable` in order to use overload (2).

Return value

(none)

Complexity

Constant.

Exceptions

If an exception is thrown, this function has no effect (strong exception guarantee).

Example

Run this code

```
#include <deque>
#include <iostream>
#include <iomanip>

int main()
{
    std::deque<std::string> numbers;

    numbers.push_back("abc");
    std::string s = "def";
    numbers.push_back(std::move(s));

    std::cout << "deque holds: ";
    for (auto&& i : numbers) std::cout << std::quoted(i) << ' ';
    std::cout << "\nMoved-from string holds " << std::quoted(s) << '\n';
}
```

Output:

```
vector holds: "abc" "def"
Moved-from string holds ""
```

See also

emplace_back (C++11)	constructs elements in-place at the end (public member function)
push_front	inserts elements to the beginning (public member function)
pop_back	removes the last element (public member function)

std::deque::emplace_back

```
template< class... Args >
void emplace_back( Args&&... args );
```
(since C++11)

Appends a new element to the end of the container. The element is constructed through `std::allocator_traits::construct`, which typically uses placement-new to construct the element in-place at the location provided by the container. The arguments `args...` are forwarded to the constructor as `std::forward<Args>(args)...`.

All iterators, including the past-the-end iterator, are invalidated. No references are invalidated.

Parameters

`args` - arguments to forward to the constructor of the element

Type requirements

- `T (the container's element type)` must meet the requirements of `EmplaceConstructible`.

Return value

(none)

Complexity

Constant.

Exceptions

If an exception is thrown, this function has no effect (strong exception guarantee).

Example

The following code uses `emplace_back` to append an object of type `President` to a `std::deque`. It demonstrates how `emplace_back` forwards parameters to the `President` constructor and shows how using `emplace_back` avoids the extra copy or move operation required when using `push_back`.

Run this code

```
#include <deque>
#include <string>
#include <iostream>

struct President
{
    std::string name;
    std::string country;
    int year;

    President(std::string p_name, std::string p_country, int p_year)
        : name(std::move(p_name)), country(std::move(p_country)), year(p_year)
    {
        std::cout << "I am being constructed.\n";
    }
    President(President&& other)
        : name(std::move(other.name)), country(std::move(other.country)), year(other.year)
    {
        std::cout << "I am being moved.\n";
    }
    President& operator=(const President& other) = default;
};
```

```
int main()
{
    std::deque<President> elections;
    std::cout << "emplace_back:\n";
    elections.emplace_back("Nelson Mandela", "South Africa", 1994);

    std::deque<President> reElections;
    std::cout << "\npush_back:\n";
    reElections.push_back(President("Franklin Delano Roosevelt", "the USA", 1936));

    std::cout << "\nContents:\n";
    for (President const& president: elections) {
        std::cout << president.name << " was elected president of "
                  << president.country << " in " << president.year << ".\n";
    }
    for (President const& president: reElections) {
        std::cout << president.name << " was re-elected president of "
                  << president.country << " in " << president.year << ".\n";
    }
}
```

Output:

```
emplace_back:
I am being constructed.

push_back:
I am being constructed.
I am being moved.

Contents:
Nelson Mandela was elected president of South Africa in 1994.
Franklin Delano Roosevelt was re-elected president of the USA in 1936.
```

See also

push_back	adds elements to the end (public member function)

std::deque::pop_back

```
void pop_back();
```

Removes the last element of the container.

Calling `pop_back` on an empty container is undefined.

Iterators and references to the erased element are invalidated. It is unspecified whether the past-the-end iterator is invalidated. Other references and iterators are not affected.	(until C++11)
Iterators and references to the erased element are invalidated. The past-the-end iterator is also invalidated. Other references and iterators are not affected.	(since C++11)

Parameters

(none)

Return value

(none)

Complexity

Constant.

Exceptions

(none)

See also

pop_front	removes the first element (public member function)
push_back	adds elements to the end (public member function)

std::deque::resize

```
void resize( size_type count, T value = T() );                          (until C++11)
void resize( size_type count );                                  (1)    (since C++11)
void resize( size_type count, const value_type& value );         (2)    (since C++11)
```

Resizes the container to contain `count` elements.

If the current size is greater than `count`, the container is reduced to its first `count` elements.

If the current size is less than `count`, additional elements are appended and initialized with copies of `value`.	(until C++11)
If the current size is less than `count`, 1) additional default-inserted elements are appended 2) additional copies of `value` are appended	(since C++11)

Parameters

`count` - new size of the container

`value` - the value to initialize the new elements with

Type requirements

- `T` must meet the requirements of `MoveInsertable` and `DefaultInsertable` in order to use overload (1).
- `T` must meet the requirements of `CopyInsertable` in order to use overload (2).

Return value

(none)

Complexity

Linear in the difference between the current size and `count`.

Example

Run this code

```
#include <iostream>
#include <deque>
int main()
{
    std::deque<int> c = {1, 2, 3};
    std::cout << "The deque holds: ";
    for(auto& el: c) std::cout << el << ' ';
    std::cout << '\n';
    c.resize(5);
    std::cout << "After resize up 5: ";
    for(auto& el: c) std::cout << el << ' ';
    std::cout << '\n';
    c.resize(2);
    std::cout << "After resize down to 2: ";
    for(auto& el: c) std::cout << el << ' ';
    std::cout << '\n';
}
```

Output:

```
The deque holds: 1 2 3
```

```
After resize up 5: 1 2 3 0 0
After resize down to 2: 1 2
```

See also

size	returns the number of elements (public member function)
insert	inserts elements (public member function)
erase	erases elements (public member function)

std::deque::swap

```
void swap( deque& other );
```

Exchanges the contents of the container with those of `other`. Does not invoke any move, copy, or swap operations on individual elements.

All iterators and references remain valid. The past-the-end iterator is invalidated.

If `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is true, then the allocators are exchanged using an unqualified call to non-member `swap`. Otherwise, they are not swapped (and if `get_allocator() != other.get_allocator()`, the behavior is undefined). (since C++11)

Parameters

other - container to exchange the contents with

Return value

(none)

Exceptions

(none) (until C++17)

`noexcept` specification: (since C++17)

```
noexcept(std::allocator_traits<Allocator>::is_always_equal::value)
```

Complexity

Constant.

See also

std::swap(std::deque)	specializes the `std::swap` algorithm (function template)

std::deque::at

```
reference       at( size_type pos );
const_reference at( size_type pos ) const;
```

Returns a reference to the element at specified location `pos`, with bounds checking.

If `pos` not within the range of the container, an exception of type `std::out_of_range` is thrown.

Parameters

pos - position of the element to return

Return value

Reference to the requested element.

Exceptions

`std::out_of_range` if `!(pos < size())`.

Complexity

Constant.

See also

operator[]	access specified element (public member function)

std::deque::operator[]

```
reference       operator[]( size_type pos );
const_reference operator[]( size_type pos ) const;
```

Returns a reference to the element at specified location `pos`. No bounds checking is performed.

Parameters

pos - position of the element to return

Return value

Reference to the requested element.

Complexity

Constant.

Notes

Unlike `std::map::operator[]`, this operator never inserts a new element into the container.

Example

The following code uses `operator[]` to read from and write to a `std::deque<int>`:

Run this code

```
#include <deque>
#include <iostream>

int main()
{
    std::deque<int> numbers {2, 4, 6, 8};

    std::cout << "Second element: " << numbers[1] << '\n';

    numbers[0] = 5;

    std::cout << "All numbers:";
    for (auto i : numbers) {
        std::cout << ' ' << i;
    }
    std::cout << '\n';
}
```

Output:

```
Second element: 4
All numbers: 5 4 6 8
```

See also

at	access specified element with bounds checking (public member function)

operator==,!=,<,<=,>,>=(std::deque)

```
template< class T, class Alloc >
bool operator==( const deque<T,Alloc>& lhs,      (1)
                 const deque<T,Alloc>& rhs );
template< class T, class Alloc >
bool operator!=( const deque<T,Alloc>& lhs,      (2)
                 const deque<T,Alloc>& rhs );
template< class T, class Alloc >
bool operator<( const deque<T,Alloc>& lhs,       (3)
                const deque<T,Alloc>& rhs );
template< class T, class Alloc >
bool operator<=( const deque<T,Alloc>& lhs,      (4)
                 const deque<T,Alloc>& rhs );
template< class T, class Alloc >
bool operator>( const deque<T,Alloc>& lhs,       (5)
                const deque<T,Alloc>& rhs );
template< class T, class Alloc >
bool operator>=( const deque<T,Alloc>& lhs,      (6)
                 const deque<T,Alloc>& rhs );
```

Compares the contents of two containers.

1-2) Checks if the contents of `lhs` and `rhs` are equal, that is, whether `lhs.size() == rhs.size()` and each element in `lhs` compares equal with the element in `rhs` at the same position.

3-6) Compares the contents of `lhs` and `rhs` lexicographically. The comparison is performed by a function equivalent to `std::lexicographical_compare`.

Parameters

`lhs, rhs` - containers whose contents to compare

- `T` must meet the requirements of `EqualityComparable` in order to use overloads (1-2).
- `T` must meet the requirements of `LessThanComparable` in order to use overloads (3-6). The ordering relation must establish total order.

Return value

1) `true` if the contents of the containers are equal, `false` otherwise
2) `true` if the contents of the containers are not equal, `false` otherwise
3) `true` if the contents of the `lhs` are lexicographically *less* than the contents of `rhs`, `false` otherwise
4) `true` if the contents of the `lhs` are lexicographically *less* than or *equal* the contents of `rhs`, `false` otherwise
5) `true` if the contents of the `lhs` are lexicographically *greater* than the contents of `rhs`, `false` otherwise
6) `true` if the contents of the `lhs` are lexicographically *greater* than or *equal* the contents of `rhs`, `false` otherwise

Complexity

Linear in the size of the container

std::swap(std::deque)

```
template< class T, class Alloc >
void swap( deque<T,Alloc>& lhs,
           deque<T,Alloc>& rhs );
```

Specializes the `std::swap` algorithm for `std::deque`. Swaps the contents of `lhs` and `rhs`. Calls `lhs.swap(rhs)`.

Parameters

`lhs, rhs` - containers whose contents to swap

Return value

(none)

Complexity

Constant.

Exceptions

(since C++17)

`noexcept` specification:

```
noexcept(noexcept(lhs.swap(rhs)))
```

See also

`swap`	swaps the contents (public member function)