deque<T, Alloc>

Containers

Type

Category: containers

Component type: type

Description

A `deque` [1] is very much like a `vector`: like `vector`, it is a sequence that supports random access to elements, constant time insertion and removal of elements at the end of the sequence, and linear time insertion and removal of elements in the middle.

The main way in which `deque` differs from `vector` is that `deque` also supports constant time insertion and removal of elements at the beginning of the sequence [2]. Additionally, `deque` does not have any member functions analogous to `vector`'s `capacity()` and `reserve()`, and does not provide any of the guarantees on iterator validity that are associated with those member functions. [3]

Example

```
deque<int> Q;
Q.push_back(3);
Q.push_front(1);
Q.insert(Q.begin() + 1, 2);
Q[2] = 0;
copy(Q.begin(), Q.end(), ostream_iterator<int>(cout, " "));
// The values that are printed are 1 2 0
```

Definition

Defined in the standard header deque, and in the nonstandard backward-compatibility header deque.h.

Template parameters

Parameter	Description	Default
`T`	The deque's value type: the type of object that is stored in the deque.	
`Alloc`	The `deque`'s allocator, used for all internal memory management.	`alloc`

Model of

Random access container, Front insertion sequence, Back insertion sequence.

Type requirements

None, except for those imposed by the requirements of Random access container, Front insertion sequence, and Back insertion sequence.

Public base classes

None.

Members

Member	Where defined	Description
`value_type`	Container	The type of object, `T`, stored in the deque.
`pointer`	Container	Pointer to `T`.
`reference`	Container	Reference to `T`
`const_reference`	Container	Const reference to `T`
`size_type`	Container	An unsigned integral type.
`difference_type`	Container	A signed integral type.
`iterator`	Container	Iterator used to iterate through a `deque`.
`const_iterator`	Container	Const iterator used to iterate through a `deque`.
`reverse_iterator`	Reversible Container	Iterator used to iterate backwards through a `deque`.
`const_reverse_iterator`	Reversible Container	Const iterator used to iterate backwards through a `deque`.
`iterator begin()`	Container	Returns an `iterator` pointing to the beginning of the `deque`.
`iterator end()`	Container	Returns an `iterator` pointing to the end of the `deque`.
`const_iterator begin() const`	Container	Returns a `const_iterator` pointing to the beginning of the `deque`.
`const_iterator end() const`	Container	Returns a `const_iterator` pointing to the end of the `deque`.
`reverse_iterator rbegin()`	Reversible Container	Returns a `reverse_iterator` pointing to the beginning of the reversed deque.
`reverse_iterator rend()`	Reversible Container	Returns a `reverse_iterator` pointing to the end of the reversed deque.
`const_reverse_iterator rbegin() const`	Reversible Container	Returns a `const_reverse_iterator` pointing to the beginning of the reversed deque.
`const_reverse_iterator rend() const`	Reversible	Returns a `const_reverse_iterator`

	Container	pointing to the end of the reversed deque.
`size_type size() const`	Container	Returns the size of the `deque`.
`size_type max_size() const`	Container	Returns the largest possible size of the `deque`.
`bool empty() const`	Container	`true` if the `deque`'s size is `0`.
`reference operator[](size_type n)`	Random Access Container	Returns the `n`'th element.
`const_reference operator[](size_type n) const`	Random Access Container	Returns the `n`'th element.
`deque()`	Container	Creates an empty deque.
`deque(size_type n)`	Sequence	Creates a deque with `n` elements.
`deque(size_type n, const T& t)`	Sequence	Creates a deque with `n` copies of `t`.
`deque(const deque&)`	Container	The copy constructor.

`template <class InputIterator>`
`deque(InputIterator f, InputIterator l)`
[4]
	Sequence	Creates a deque with a copy of a range.

`~deque()`	Container	The destructor.
`deque& operator=(const deque&)`	Container	The assignment operator
`reference front()`	Front Insertion Sequence	Returns the first element.
`const_reference front() const`	Front Insertion Sequence	Returns the first element.
`reference back()`	Back Insertion Sequence	Returns the last element.
`const_reference back() const`	Back Insertion Sequence	Returns the last element.
`void push_front(const T&)`	Front Insertion Sequence	Inserts a new element at the beginning.
`void push_back(const T&)`	Back Insertion Sequence	Inserts a new element at the end.
`void pop_front()`	Front Insertion Sequence	Removes the first element.
`void pop_back()`	Back Insertion	Removes the last element.

	Sequence	
`void swap(deque&)`	Container	Swaps the contents of two deques.
`iterator insert(iterator pos, const T& x)`	Sequence	Inserts `x` before `pos`.
`template <class InputIterator> void insert(iterator pos, InputIterator f, InputIterator l)` [4]	Sequence	Inserts the range `[f, l)` before `pos`.
`void insert(iterator pos, size_type n, const T& x)`	Sequence	Inserts `n` copies of `x` before `pos`.
`iterator erase(iterator pos)`	Sequence	Erases the element at position `pos`.
`iterator erase(iterator first, iterator last)`	Sequence	Erases the range `[first, last)`
`void clear()`	Sequence	Erases all of the elements.
`void resize(n, t = T())`	Sequence	Inserts or erases elements at the end such that the size becomes `n`.
`bool operator==(const deque&, const deque&)`	Forward Container	Tests two deques for equality. This is a global function, not a member function.
`bool operator<(const deque&, const deque&)`	Forward Container	Lexicographical comparison. This is a global function, not a member function.

New members

All of `deque`'s members are defined in the Random access container, Front insertion sequence, and Back insertion sequence requirements. `Deque` does not introduce any new members.

Notes

[1] The name *deque* is pronounced "deck", and stands for "double-ended queue." Knuth (section 2.6) reports that the name was coined by E. J. Schweppe. See section 2.2.1 of Knuth for more information about deques. (D. E. Knuth, *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, second edition. Addison-Wesley, 1973.)

[2] Inserting an element at the beginning or end of a `deque` takes amortized constant time. Inserting an element in the middle is linear in `n`, where `n` is the smaller of the number of elements from the insertion point to the beginning, and the number of elements from the insertion point to the end.

[3] The semantics of iterator invalidation for `deque` is as follows. `Insert` (including `push_front` and `push_back`) invalidates all iterators that refer to a `deque`. `Erase` in the middle of a `deque` invalidates all iterators that refer to the `deque`. `Erase` at the beginning or end of a `deque` (including `pop_front` and `pop_back`) invalidates an iterator only if it points to the erased element.

[4] This member function relies on *member template* functions, which at present (early 1998) are not supported by all compilers. If your compiler supports member templates, you can call this function with any type of input iterator. If your compiler does not yet support member templates, though, then the arguments must either be of type `const value_type*` or of type `deque::const_iterator`.

See also

`vector`, `list`, `slist`
