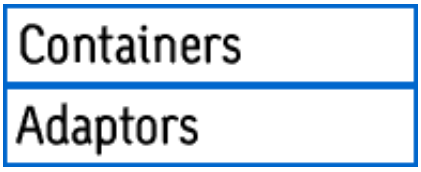
queue<T, Sequence>



Categories: containers, adaptors

Component type: type

Description

A `queue` is an adaptor that provides a restricted subset of Container functionality A `queue` is a "first in first out" (FIFO) data structure. [1] That is, elements are added to the back of the `queue` and may be removed from the front; `Q.front()` is the element that was added to the `queue` least recently. `Queue` does not allow iteration through its elements. [2]

`Queue` is a container adaptor, meaning that it is implemented on top of some underlying container type. By default that underlying type is `deque`, but a different type may be selected explicitly.

Example

```
int main() {
  queue<int> Q;
  Q.push(8);
  Q.push(7);
  Q.push(6);
  Q.push(2);

  assert(Q.size() == 4);
  assert(Q.back() == 2);

  assert(Q.front() == 8);
  Q.pop();

  assert(Q.front() == 7);
  Q.pop();

  assert(Q.front() == 6);
  Q.pop();

  assert(Q.front() == 2);
  Q.pop();

  assert(Q.empty());
}
```

Definition

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

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the queue.	
`Sequence`	The type of the underlying container used to implement the queue.	`deque<T>`

Model of

Assignable, Default Constructible

Type requirements

- `T` is a model of Assignable.
- `Sequence` is a model of Front Insertion Sequence.
- `Sequence` is a model of Back Insertion Sequence.
- `Sequence::value_type` is the same type as `T`.
- If `operator==` is used, then `T` is a model of Equality Comparable
- If `operator<` is used, then `T` is a model of LessThan Comparable.

Public base classes

None.

Members

Member	Where defined	Description
`value_type`	`queue`	See below.
`size_type`	`queue`	See below.
`queue()`	Default Constructible	The default constructor. Creates an empty `queue`.
`queue(const queue&)`	Assignable	The copy constructor.
`queue& operator=(const queue&)`	Assignable	The assignment operator.
`bool empty() const`	`queue`	See below.
`size_type size() const`	`queue`	See below.
`value_type& front()`	`queue`	See below.
`const value_type& front() const`	`queue`	See below.
`value_type& back()`	`queue`	See below.
`const value_type& back() const`	`queue`	See below.
`void push(const value_type&)`	`queue`	See below.

`void pop()` [3]	`queue`	See below.
`bool operator==(const queue&, const queue&)`	`queue`	See below.
`bool operator<(const queue&, const queue&)`	`queue`	See below.

New members

These members are not defined in the Assignable and Default Constructible requirements, but are specific to `queue`.

Member	Description
`value_type`	The type of object stored in the `queue`. This is the same as `T` and `Sequence::value_type`.
`size_type`	An unsigned integral type. This is the same as `Sequence::size_type`.
`bool empty() const`	Returns `true` if the `queue` contains no elements, and `false` otherwise. `Q.empty()` is equivalent to `Q.size() == 0`.
`size_type size() const`	Returns the number of elements contained in the `queue`.
`value_type& front()`	Returns a mutable reference to the element at the front of the queue, that is, the element least recently inserted. Precondition: `empty()` is `false`.
`const value_type& front() const`	Returns a const reference to the element at the front of the queue, that is, the element least recently inserted. Precondition: `empty()` is `false`.
`value_type& back()`	Returns a mutable reference to the element at the back of the queue, that is, the element most recently inserted. Precondition: `empty()` is `false`.
`const value_type& back() const`	Returns a const reference to the element at the back of the queue, that is, the element most recently inserted. Precondition: `empty()` is `false`.
`void push(const value_type& x)`	Inserts `x` at the back of the queue. Postconditions: `size()` will be incremented by `1`, and `back()` will be equal to `x`.
`void pop()`	Removes the element at the front of the queue. [3] Precondition: `empty()` is `false`. Postcondition: `size()` will be decremented by `1`.
`bool operator== (const queue&, const queue&)`	Compares two queues for equality. Two queues are equal if they contain the same number of elements and if they are equal element-by-element. This is a global function, not a member function.
`bool operator<(const queue&, const queue&)`	Lexicographical ordering of two queues. This is a global function, not a member function.

Notes

[1] Queues are a standard data structure, and are discussed in all algorithm books. See, for example, section 2.2.1 of Knuth. (D. E. Knuth, *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, second edition. Addison-Wesley, 1973.)

[2] This restriction is the only reason for `queue` to exist at all. Any container that is both a front insertion sequence and a back insertion sequence can be used as a queue; `deque`, for example, has member

functions `front`, `back`, `push_front`, `push_back`, `pop_front`, and `pop_back` The only reason to use the container adaptor `queue` instead of the container `deque` is to make it clear that you are performing only queue operations, and no other operations.

[3] One might wonder why `pop()` returns `void`, instead of `value_type`. That is, why must one use `front()` and `pop()` to examine and remove the element at the front of the `queue`, instead of combining the two in a single member function? In fact, there is a good reason for this design. If `pop()` returned the front element, it would have to return by value rather than by reference: return by reference would create a dangling pointer. Return by value, however, is inefficient: it involves at least one redundant copy constructor call. Since it is impossible for `pop()` to return a value in such a way as to be both efficient and correct, it is more sensible for it to return no value at all and to require clients to use `front()` to inspect the value at the front of the `queue`.

See also

`stack`, `priority_queue`, `deque`, Container, Sequence
