std::priority_queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
> class priority_queue;
```

A priority queue is a container adaptor that provides constant time lookup of the largest (by default) element, at the expense of logarithmic insertion and extraction.

A user-provided `Compare` can be supplied to change the ordering, e.g. using `std::greater<T>` would cause the smallest element to appear as the `top()`.

Working with a `priority_queue` is similar to managing a heap in some random access container, with the benefit of not being able to accidentally invalidate the heap.

Template parameters

`T` - The type of the stored elements.

`Container` - The type of the underlying container to use to store the elements. The container must satisfy the requirements of `SequenceContainer`. Additionally, it must provide the following functions with the usual semantics:

- `front()`
- `push_back()`
- `pop_back()`

The standard containers `std::vector` and `std::deque` satisfy these requirements.

`Compare` - A `Compare` type providing a strict weak ordering.

Member types

Member type	Definition
`container_type`	`Container`
`value_type`	`Container::value_type`
`size_type`	`Container::size_type`
`reference`	`Container::reference`
`const_reference`	`Container::const_reference`

Member functions

(constructor)	constructs the `priority_queue` (public member function)
(destructor)	destructs the `priority_queue` (public member function)
`operator=`	assigns values to the container adaptor (public member function)

Element access

`top`	accesses the top element (public member function)

Capacity

empty	checks whether the underlying container is empty (public member function)
size	returns the number of elements (public member function)

Modifiers

push	inserts element and sorts the underlying container (public member function)
emplace (C++11)	constructs element in-place and sorts the underlying container (public member function)
pop	removes the top element (public member function)
swap	swaps the contents (public member function)

Member objects

`Container` **c**	the underlying container (protected member object)
`Compare` **comp**	the comparison function object (protected member object)

Non-member functions

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

Helper classes

std::uses_allocator<std::priority_queue> (C++11)	specializes the `std::uses_allocator` type trait (function template)

Example

Run this code

```
#include <functional>
#include <queue>
#include <vector>
#include <iostream>

template<typename T> void print_queue(T& q) {
    while(!q.empty()) {
        std::cout << q.top() << " ";
        q.pop();
    }
    std::cout << '\n';
}

int main() {
    std::priority_queue<int> q;

    for(int n : {1,8,5,6,3,4,0,9,3,2})
        q.push(n);

    print_queue(q);

    std::priority_queue<int, std::vector<int>, std::greater<int> > q2;

    for(int n : {1,8,5,6,3,4,0,9,3,2})
```

```
        q2.push(n);

    print_queue(q2);
}
```

Output:

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

std::priority_queue::priority_queue

`explicit priority_queue( const Compare& compare = Compare(), const Container& cont = Container() );`	(1)	(until C++11)
`priority_queue( const Compare& compare, const Container& cont );`		(since C++11)
`explicit priority_queue( const Compare& compare = Compare(), Container&& cont = Container() );`	(2)	(since C++11)
`priority_queue( const priority_queue& other );`	(3)	
`priority_queue( priority_queue&& other );`	(4)	(since C++11)
`template< class Alloc > explicit priority_queue( const Alloc& alloc );`	(5)	(since C++11)
`template< class Alloc > priority_queue( const Compare& compare, const Alloc& alloc );`	(6)	(since C++11)
`template< class Alloc > priority_queue( const Compare& compare, const Container& cont, const Alloc& alloc );`	(7)	(since C++11)
`template< class Alloc > priority_queue( const Compare& compare, Container&& cont, const Alloc& alloc );`	(8)	(since C++11)
`template< class Alloc > priority_queue( const priority_queue& other, const Alloc& alloc );`	(9)	(since C++11)
`template< class Alloc > priority_queue( priority_queue&& other, const Alloc& alloc );`	(10)	(since C++11)
`template< class InputIt > priority_queue( InputIt first, InputIt last, const Compare& compare, const Container& cont );`	(11)	(since C++11)
`template< class InputIt > priority_queue( InputIt first, InputIt last, const Compare& compare = Compare(), Container&& cont = Container() );`	(12)	(since C++11)

Constructs new underlying container of the container adaptor from a variety of data sources.

1) Copy-constructs the underlying container `c` with the contents of `cont`. Copy-constructs the comparison functor `comp` with the contents of `compare`. Calls `std::make_heap(c.begin(), c.end(), comp)`. This is also the default constructor. (until C++11)
2) Move-constructs the underlying container `c` with `std::move(cont)`. Move-constructs the comparison functor `comp` with `std::move(compare)`. Calls `std::make_heap(c.begin(), c.end(), comp)`. This is also the default constructor. (since C++11)
3) Copy constructor. The adaptor is copy-constructed with the contents of `other.c`. The comparison functor is constructed with `std::move(other.comp)`. (implicitly declared)
4) Move constructor. The adaptor is constructed with `std::move(other.c)`.The comparison functor is constructed with `std::move(other.comp)`. (implicitly declared)

5-10) The following constructors are only defined if `std::uses_allocator<container_type, Alloc>::value == true`, that is, if the underlying container is an allocator-aware container (true for all standard library containers).

5) Constructs the underlying container using `alloc` as allocator. Effectively calls `c(alloc)`. `comp` is value-initialized.
6) Constructs the underlying container using `alloc` as allocator. Effectively calls `c(alloc)`. Copy-constructs `comp` from `compare`.
7) Constructs the underlying container with the contents of `cont` and using `alloc` as allocator. Effectively calls `c(cont, alloc)`. Copy-constructs `comp` from `compare`.
8) Constructs the underlying container with the contents of `cont` using move semantics while

utilising `alloc` as allocator. Effectively calls `c(std::move(cont), alloc)`. Copy-constructs `comp` from `compare`.

9) Constructs the adaptor with the contents of `other.c` and using `alloc` as allocator. Effectively calls `c(athor.c, alloc)`. Copy-constructs `comp` from `other.comp`.

10) Constructs the adaptor with the contents of `other` using move semantics while utilising `alloc` as allocator. Effectively calls `c(std::move(other.c), alloc)`. Move-constructs `comp` from `other.comp`.

11) Copy-constructs `c` from `cont` and `comp` from `compare`. Then calls `c.insert(c.end(), first, last);`, and then calls `std::make_heap(c.begin(), c.end(), comp);`.

12) Move-constructs `c` from `std::move(cont)` and `comp` from `std::move(compare)`. Then calls `c.insert(c.end(), first, last);`, and then calls `std::make_heap(c.begin(), c.end(), comp);`.

Parameters

alloc	-	allocator to use for all memory allocations of the underlying container
other	-	another container adaptor to be used as source to initialize the underlying container
cont	-	container to be used as source to initialize the underlying container
compare	-	the comparison function object to initialize the underlying comparison functor
first, last	-	range of elements to initialize with

Type requirements

- `Alloc` must meet the requirements of `Allocator`.
- `Container` must meet the requirements of `Container`. The constructors (5-10) are only defined if `Container` meets the requirements of `AllocatorAwareContainer`
- `InputIt` must meet the requirements of `InputIterator`.

Complexity

1,3) $O(N)$ comparisons, where N is `cont.size()`.
Additionally, $O(N)$ calls to the constructor of `value_type`, where N is `cont.size()`.

2) $O(N)$ comparisons, where N is `cont.size()`.

4-6) Constant.

7) $O(N)$ comparisons, where N is `cont.size()`.
Additionally, $O(N)$ calls to the constructor of `value_type`, where N is `cont.size()`.

8) $O(N)$ comparisons, where N is `cont.size()`.

9) Linear in size of `other`.

10) Constant.

11) $O(N)$ comparisons, where N is `cont.size() + std::distance(first, last)`.
Additionally, $O(N)$ calls to the constructor of `value_type`, where N is `cont.size()`.

12) $O(N)$ comparisons, where N is `cont.size() + std::distance(first, last)`.

Example

Run this code

```
#include <queue>
#include <vector>
#include <iostream>
#include <functional>

int main()
{
    std::priority_queue<int> c1;
```

```
    c1.push(5);
    std::cout << c1.size() << '\n';

    std::priority_queue<int> c2(c1);
    std::cout << c2.size() << '\n';

    std::vector<int> vec={3, 1, 4, 1, 5};
    std::priority_queue<int> c3(std::less<int>(), vec);
    std::cout << c3.size() << '\n';
}
```

Output:

```
1
1
5
```

See also

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

std::priority_queue::~priority_queue

```
~priority_queue();
```

Destructs the container adaptor. 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 adaptor.

std::priority_queue::operator=

```
priority_queue operator=( const priority_queue& other );     (1)
priority_queue operator=( priority_queue&& other );          (2)  (since C++11)
```

Replaces the contents of the container adaptor with those of `other`.

1) Copy assignment operator. Replaces the contents with a copy of the contents of `other`. Effectively calls `c = other.c;` . (implicitly declared)
2) Move assignment operator. Replaces the contents with those of `other` using move semantics. Effectively calls `c = std::move(other.c);` (implicitly declared)

Parameters

`other` - another container adaptor to be used as source

Return value

`*this`

Complexity

Equivalent to that of `operator=` of the underlying container.

See also

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

std::priority_queue::top

```
const_reference top() const;
```

Returns reference to the top element in the priority queue. This element will be removed on a call to `pop()`. If default comparison function is used, the returned element is also the greatest among the elements in the queue.

Parameters

(none)

Return value

Reference to the top element as if obtained by a call to `c.front()`

Complexity

Constant.

See also

pop	removes the top element (public member function)

std::priority_queue::empty

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

Checks if the underlying container has no elements, i.e. whether `c.empty()`.

Parameters

(none)

Return value

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

Complexity

Constant

See also

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

std::priority_queue::size

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

Returns the number of elements in the underlying container, that is, `c.size()`.

Parameters

(none)

Return value

The number of elements in the container.

Complexity

Constant.

See also

empty	checks whether the underlying container is empty (public member function)

std::priority_queue::push

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

Pushes the given element `value` to the priority queue.

1) Effectively calls `c.push_back(value); std::push_heap(c.begin(), c.end(), comp);`
2) Effectively calls `c.push_back(std::move(value)); std::push_heap(c.begin(), c.end(), comp);`

Parameters

value - the value of the element to push

Return value

(none)

Complexity

Logarithmic number of comparisons plus the complexity of `Container::push_back`.

See also

emplace (C++11)	constructs element in-place and sorts the underlying container (public member function)
pop	removes the top element (public member function)

std::priority_queue::emplace

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

Pushes new element to the priority queue. The element is constructed in-place, i.e. no copy or move operations are performed. The constructor of the element is called with exactly the same arguments as supplied to the function.

Effectively calls
`c.emplace_back(std::forward<Args>(args)...); std::push_heap(c.begin(), c.end(), comp);`

Parameters

args - arguments to forward to the constructor of the element

Return value

(none)

Complexity

Logarithmic number of comparisons plus the complexity of `Container::emplace_back`.

See also

push	inserts element and sorts the underlying container (public member function)
pop	removes the top element (public member function)

std::priority_queue::pop

```
void pop();
```

Removes the top element from the priority queue. Effectively calls `std::pop_heap(c.begin(), c.end(), comp); c.pop_back();`

Parameters

(none)

Return value

(none)

Complexity

Logarithmic number of comparisons plus the complexity of `Container::pop_back`.

See also

emplace (C++11)	constructs element in-place and sorts the underlying container (public member function)
push	inserts element and sorts the underlying container (public member function)

std::priority_queue::swap

```
void swap( priority_queue& other );    (since C++11)
```

Exchanges the contents of the container adaptor with those of `other`. Effectively calls `using std::swap; swap(c, other.c); swap(comp, other.comp);`

Parameters

other - container adaptor to exchange the contents with

Return value

(none)

Exceptions

`noexcept` specification:

```
noexcept(noexcept(std::swap(c, other.c)) && noexcept(std::swap(comp, other.comp)))
```

Complexity

Same as underlying container (typically constant)

See also

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

std::swap(std::priority_queue)

```
template< class T, class Container, class Compare >
void swap( priority_queue<T,Container,Compare>& lhs,
           priority_queue<T,Container,Compare>& rhs );
```

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

Parameters

`lhs, rhs` - containers whose contents to swap

Return value

(none)

Complexity

Same as swapping the underlying container.

Exceptions

(since C++17)

`noexcept` specification:

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

See also

`swap`	swaps the contents (public member function)

std::uses_allocator<std::priority_queue>

```
template< class T, class Container, class Compare,class Alloc >
struct uses_allocator<priority_queue<T,Compare,Container>,Alloc> :
    std::uses_allocator<Container, Alloc>::type { };
```
(since C++11)

Provides a transparent specialization of the `std::uses_allocator` type trait for `std::priority_queue` : the container adaptor uses allocator if and only if the underlying container does.

Inherited from std::integral_constant

Member constants

value [static]	`true` (public static member constant)

Member functions

operator bool	converts the object to `bool` , returns `value` (public member function)
operator() (C++14)	returns `value` (public member function)

Member types

Type	Definition
`value_type`	`bool`
`type`	`std::integral_constant<bool, value>`

See also

uses_allocator (C++11)	checks if the specified type supports uses-allocator construction (class template)