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;
```

The 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. The behavior is undefined if T is not the same type as `Container::value_type`.

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

- `front()`, e.g., `std::vector::front()`,
- `push_back()`, e.g., `std::deque::push_back()`,
- `pop_back()`, e.g., `std::vector::pop_back()`.

The standard containers `std::vector` (including `std::vector<bool>`) and `std::deque` satisfy these requirements.

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

Note that the *Compare* parameter is defined such that it returns `true` if its first argument comes *before* its second argument in a weak ordering. But because the priority queue outputs largest elements first, the elements that "come before" are actually output last. That is, the front of the queue contains the "last" element according to the weak ordering imposed by *Compare*.

Member types

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

Member objects

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

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 container adaptor 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)
`push_range` (C++23)	inserts a range of elements 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` (C++11)	swaps the contents (public member function)

Non-member functions

`std::swap``(std::priority_queue)` (C++11)	specializes the `std::swap` algorithm (function template)

Helper classes

`std::uses_allocator``<std::priority_queue>` (C++11)	specializes the `std::uses_allocator` type trait (class template specialization)
`std::formatter``<std::priority_queue>` (C++23)	formatting support for `std::priority_queue` (class template specialization)

Deduction guides (since C++17)

Notes

Feature-test macro	Value	Std	Feature
`__cpp_lib_containers_ranges`	`202202L`	(C++23)	Ranges-aware construction and insertion for containers

Example

Run this code

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

```
template<typename T>
void pop_println(std::string_view rem, T& pq)
{
    std::cout << rem << ": ";
    for (; !pq.empty(); pq.pop())
        std::cout << pq.top() << ' ';
    std::cout << '\n';
}

template<typename T>
void println(std::string_view rem, const T& v)
{
    std::cout << rem << ": ";
    for (const auto& e : v)
        std::cout << e << ' ';
    std::cout << '\n';
}

int main()
{
    const auto data = {1, 8, 5, 6, 3, 4, 0, 9, 7, 2};
    println("data", data);

    std::priority_queue<int> max_priority_queue;

    // Fill the priority queue.
    for (int n : data)
        max_priority_queue.push(n);

    pop_println("max_priority_queue", max_priority_queue);

    // std::greater<int> makes the max priority queue act as a min priority queue.
    std::priority_queue<int, std::vector<int>, std::greater<int>>
        min_priority_queue1(data.begin(), data.end());

    pop_println("min_priority_queue1", min_priority_queue1);

    // Second way to define a min priority queue.
    std::priority_queue min_priority_queue2(data.begin(), data.end(), std::greater<int>());

    pop_println("min_priority_queue2", min_priority_queue2);

    // Using a custom function object to compare elements.
    struct
    {
        bool operator()(const int l, const int r) const { return l > r; }
    } customLess;

    std::priority_queue custom_priority_queue(data.begin(), data.end(), customLess);

    pop_println("custom_priority_queue", custom_priority_queue);

    // Using lambda to compare elements.
    auto cmp = [](int left, int right) { return (left ^ 1) < (right ^ 1); };
    std::priority_queue<int, std::vector<int>, decltype(cmp)> lambda_priority_queue(cmp);

    for (int n : data)
        lambda_priority_queue.push(n);

    pop_println("lambda_priority_queue", lambda_priority_queue);
}
```

Output:

```
data: 1 8 5 6 3 4 0 9 7 2
max_priority_queue: 9 8 7 6 5 4 3 2 1 0
min_priority_queue1: 0 1 2 3 4 5 6 7 8 9
min_priority_queue2: 0 1 2 3 4 5 6 7 8 9
custom_priority_queue: 0 1 2 3 4 5 6 7 8 9
lambda_priority_queue: 8 9 6 7 4 5 2 3 0 1
```

Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

DR	Applied to	Behavior as published	Correct behavior
LWG 307 (https://cplusplus.github.io/LWG/issue307)	C++98	`Container` could not be `std::vector<bool>`	allowed
LWG 2684 (https://cplusplus.github.io/LWG/issue2684)	C++98	`priority_queue` takes a comparator but lacked member typedef for it	added

See also

`vector`	dynamic contiguous array (class template)
`vector<bool>`	space-efficient dynamic bitset (class template specialization)
`deque`	double-ended queue (class template)