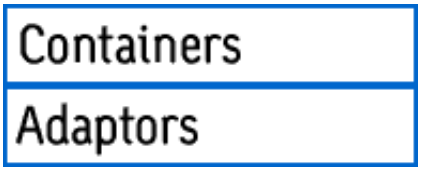
priority_queue<T, Sequence, Compare>



Categories: containers, adaptors

Component type: type

Description

A `priority_queue` is an adaptor that provides a restricted subset of Container functionality: it provides insertion of elements, and inspection and removal of the top element. It is guaranteed that the top element is the largest element in the `priority_queue`, where the function object `Compare` is used for comparisons. [1] `Priority_queue` does not allow iteration through its elements. [2]

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

Example

```
int main() {
  priority_queue<int> Q;
  Q.push(1);
  Q.push(4);
  Q.push(2);
  Q.push(8);
  Q.push(5);
  Q.push(7);

  assert(Q.size() == 6);

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

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

  assert(Q.top() == 5);
  Q.pop();

  assert(Q.top() == 4);
  Q.pop();

  assert(Q.top() == 2);
  Q.pop();
```

```
  assert(Q.top() == 1);
  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 priority queue.	
`Sequence`	The type of the underlying container used to implement the priority queue.	`vector<T>`
`Compare`	The comparison function used to determine whether one element is smaller than another element. If `Compare(x,y)` is `true`, then `x` is smaller than `y`. The element returned by `Q.top()` is the largest element in the priority queue. That is, it has the property that, for every other element `x` in the priority queue, `Compare(Q.top(), x)` is `false`.	`less<T>`

Model of

Assignable, Default Constructible

Type requirements

- `T` is a model of Assignable.
- `Sequence` is a model of Sequence.
- `Sequence` is a model of Random Access Container
- `Sequence::value_type` is the same type as `T`.
- `Compare` is a model of Binary Predicate
- `Compare` induces a strict weak ordering, as defined in the LessThan Comparable requirements, on its argument type.
- `T` is convertible to `Compare`'s argument type.

Public base classes

None.

Members

Member	Where defined	Description
`value_type`	`priority_queue`	See below.
`size_type`	`priority_queue`	See below.
`priority_queue()`	Default	The default constructor. Creates an empty

	Constructible	`priority_queue`, using `Compare()` as the comparison function.
`priority_queue(const priority_queue&)`	Assignable	The copy constructor.
`priority_queue(const Compare&)`	`priority_queue`	See below.
`priority_queue(const value_type*, const value_type*)`	`priority_queue`	See below.
`priority_queue(const value_type*, const value_type*, const Compare&)`	`priority_queue`	See below.
`priority_queue& operator=(const priority_queue&)`	Assignable	The assignment operator.
`bool empty() const`	`priority_queue`	See below.
`size_type size() const`	`priority_queue`	See below.
`const value_type& top() const`	`priority_queue`	See below.
`void push(const value_type&)`	`priority_queue`	See below.
`void pop()` [3]	`priority_queue`	See below.

New members

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

Member	Description
`value_type`	The type of object stored in the `priority_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`.
`priority_queue(const Compare& comp)`	The constructor. Creates an empty `priority_queue`, using `comp` as the comparison function. The default constructor uses `Compare()` as the comparison function.
`priority_queue(const value_type* first, const value_type* last)`	The constructor. Creates a `priority_queue` initialized to contain the elements in the range `[first, last)`, and using `Compare()` as the comparison function.
`priority_queue(const value_type* first, const value_type* last, const Compare& comp)`	The constructor. Creates a `priority_queue` initialized to contain the elements in the range `[first, last)`, and using `comp` as the comparison function.
`bool empty() const`	Returns `true` if the `priority_queue` contains no elements, and `false` otherwise. `S.empty()` is equivalent to `S.size() == 0`.
`size_type size() const`	Returns the number of elements contained in the `priority_queue`.
`const value_type& top() const`	Returns a const reference to the element at the top of the priority_queue. The element at the top is guaranteed to be the largest element in the priority queue, as determined by the comparison function `Compare`. That is, for every other

	element x in the priority_queue, Compare(Q.top(), x) is false. Precondition: empty() is false.
void push(const value_type& x)	Inserts x into the priority_queue. Postcondition: size() will be incremented by 1.
void pop()	Removes the element at the top of the priority_queue, that is, the largest element in the priority_queue. [3] Precondition: empty() is false. Postcondition: size() will be decremented by 1.

Notes

[1] Priority queues are a standard concept, and can be implemented in many different ways; this implementation uses heaps. Priority queues are discussed in all algorithm books; see, for example, section 5.2.3 of Knuth. (D. E. Knuth, *The Art of Computer Programming. Volume 3: Sorting and Searching*. Addison-Wesley, 1975.)

[2] This restriction is the only reason for priority_queue to exist at all. If iteration through elements is important, you can either use a vector that is maintained in sorted order, or a set, or a vector that is maintained as a heap using make_heap, push_heap, and pop_heap. Priority_queue is, in fact, implemented as a random access container that is maintained as a heap. The only reason to use the container adaptor priority_queue, instead of performing the heap operations manually, is to make it clear that you are never performing any operations that might violate the heap invariant.

[3] One might wonder why pop() returns void, instead of value_type. That is, why must one use top() and pop() to examine and remove the element at the top of the priority_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 top 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 top() to inspect the value at the top of the priority_queue.

See also

stack, queue, set, make_heap, push_heap, pop_heap, is_heap, sort, is_sorted, Container, Sorted Associative Container, Sequence

---