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

Priority queue

Priority queues are a type of container adaptors, specifically designed such that its first element is always the greatest of the elements it contains, according to some *strict weak ordering* criterion.

This context is similar to a *heap*, where elements can be inserted at any moment, and only the *max heap* element can be retrieved (the one at the top in the *priority queue*).

Priority queues are implemented as *container adaptors*, which are classes that use an encapsulated object of a specific container class as its *underlying container*, providing a specific set of member functions to access its elements. Elements are *popped* from the *"back"* of the specific container, which is known as the *top* of the priority queue.

The underlying container may be any of the standard container class templates or some other specifically designed container class. The container shall be accessible through *random access iterators* and support the following operations:

- `empty()`
- `size()`
- `front()`
- `push_back()`
- `pop_back()`

The standard container classes vector and deque fulfill these requirements. By default, if no container class is specified for a particular priority_queue class instantiation, the standard container vector is used.

Support of *random access iterators* is required to keep a heap structure internally at all times. This is done automatically by the container adaptor by automatically calling the algorithm functions make_heap, push_heap and pop_heap when needed.

```
template <class T, class Container = deque<T> > class queue;
```

FIFO queue

queues are a type of container adaptor, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other.

queues are implemented as *containers adaptors*, which are classes that use an encapsulated object of a specific container class as its *underlying container*, providing a specific set of member functions to access its elements. Elements are *pushed* into the *"back"* of the specific container and *popped* from its *"front"*.

The underlying container may be one of the standard container class template or some other specifically designed container class. This underlying container shall support at least the following operations:

- `empty`
- `size`
- `front`
- `back`
- `push_back`
- `pop_front`

The standard container classes `deque` and `list` fulfill these requirements. By default, if no container class is specified for a particular `queue` class instantiation, the standard container `deque` is used.

Heap

default (1)
```
template <class RandomAccessIterator>
  void make_heap (RandomAccessIterator first, RandomAccessIterator last);
```

custom (2)
```
template <class RandomAccessIterator, class Compare>
  void make_heap (RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp );
```

Make heap from range

Rearranges the elements in the range `[first,last)` in such a way that they form a *heap*.

A *heap* is a way to organize the elements of a range that allows for fast retrieval of the element with the highest value at any moment (with pop_heap), even repeatedly, while allowing for fast insertion of new elements (with push_heap).

The element with the highest value is always pointed by *first*. The order of the other elements depends on the particular implementation, but it is consistent throughout all heap-related functions of this header.

The elements are compared using `operator<` (for the first version), or *comp* (for the second): The element with the highest value is an element for which this would return `false` when compared to every other element in the range.

The standard container
adaptor `priority_queue` calls `make_heap`, `push_heap` and `pop_heap` automatically to maintain *heap properties* for a container.