
12.3. Priority Queues

The class `priority_queue<>` implements a queue from which elements are read according to their priority. The interface is similar to queues. That is, `push()` inserts an element into the queue, whereas `top()` and `pop()` access and remove the next element (Figure 12.5). However, the next element is not the first inserted element. Rather, it is the element that has the highest priority. Thus, elements are partially sorted according to their value. As usual, you can provide the sorting criterion as a template parameter. By default, the elements are sorted by using operator `<` in descending order. Thus, the next element is always the "highest" element. If more than one "highest" element exists, which element comes next is undefined.

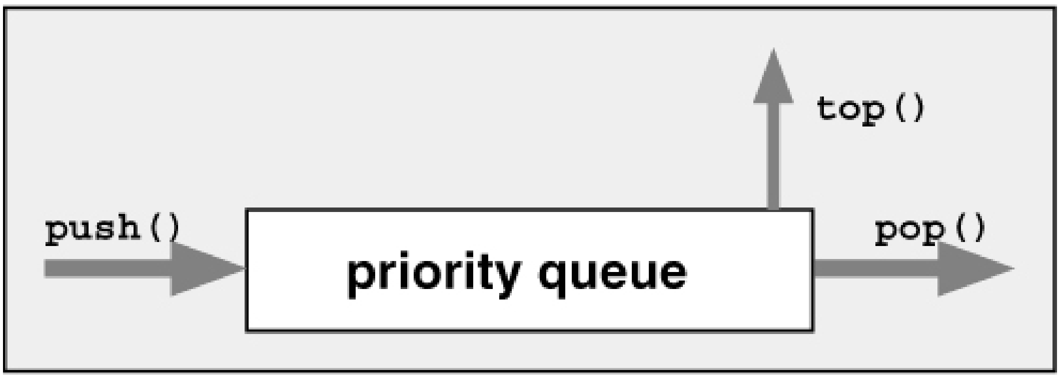

Figure 12.5. Interface of a Priority Queue

Priority queues are defined in the same header file as ordinary queues, `<queue>` :

```
#include <queue>
```

In `<queue>` , the class `priority_queue` is defined as follows:

Click here to view code image

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

The first template parameter is the type of the elements. The optional second template parameter defines the container that the priority queue uses internally for its elements. The default container is a vector. The optional third template parameter defines the sorting criterion used to find the next element with the highest priority. By default, this parameter compares the elements by using operator <. For example, the following declaration defines a priority queue of `float` s:

```
std::priority_queue<float> pbuffer;          // priority queue for floats
```

The priority queue implementation simply maps the operations into appropriate calls of the container that is used internally. You can use any sequence container class that provides random-access iterators and the member functions `front()` , `push_back()` , and `pop_back()` . Random access is necessary for sorting the elements, which is performed by the heap algorithms of the STL (see Section 11.9.4, page 604). For example, you could also use a deque as the container for the elements:

```
std::priority_queue<float,std::deque<float>> pbuffer;
```

To define your own sorting criterion, you must pass a function, a function object, or a lambda as a binary predicate that is used by the sorting algorithms to compare two elements (for more about sorting criteria, see Section 7.7.2, page 316, and Section 10.1.1, page 476). For example, the following declaration defines a priority queue with reverse sorting:

```
std::priority_queue<float,std::vector<float>,
                          std::greater<float>> pbuffer;
```

In this priority queue, the next element is always one of the elements with the lowest value.

12.3.1. The Core Interface

The core interface of priority queues is provided by the member functions `push()` , `top()` , and `pop()` :

- `push()` inserts an element into the priority queue.
- `top()` returns the next available element in the priority queue.
- `pop()` removes an element from the priority queue.

As for the other container adapters, `pop()` removes the next element but does not return it, whereas `top()` returns the next element without removing it. Thus, you must always call both functions to process and remove the next element from the priority queue. And, as usual, the behavior of `top()` and `pop()` is undefined if the priority queue contains no elements. If in doubt, you must use the member functions `size()` and `empty()` .

12.3.2. Example of Using Priority Queues

The following program demonstrates the use of class `priority_queue<>` :

Click here to view code image

```
// contadapt/priorityqueue1.cpp

#include <iostream>
#include <queue>
using namespace std;

int main()
{
    priority_queue<float> q;

    // insert three elements into the priority queue
    q.push(66.6);
    q.push(22.2);
    q.push(44.4);

    // read and print two elements
    cout << q.top() << ' ';
    q.pop();
    cout << q.top() << endl;
    q.pop();

    // insert three more elements
    q.push(11.1);
    q.push(55.5);
    q.push(33.3);

    // skip one element
    q.pop();

    // pop and print remaining elements
    while (!q.empty()) {
        cout << q.top() << ' ';
        q.pop();
    }
    cout << endl;
}
```

The output of the program is as follows:

```
66.6 44.4
33.3 22.2 11.1
```

As you can see, after `66.6` , `22.2` , and `44.4` are inserted, the program prints `66.6` and `44.4` as the highest elements. After three other elements are inserted, the priority queue contains the elements `22.2` , `11.1` , `55.5` , and `33.3` (in the order of insertion). The next element is skipped simply via a call of `pop()` , so the final loop prints `33.3` , `22.2` , and `11.1` in that order.

12.3.3. Class `priority_queue<>` in Detail

The priority queue uses the STL's heap algorithms:

Click here to view code image

```
namespace std {
   template <typename T, typename Container = vector<T>,
             typename Compare = less<typename Container::value_type>>
   class priority_queue {
     protected:
       Compare comp;      // sorting criterion
```

```
          Container c;        // container
        public:
          // constructors
          explicit priority_queue(const Compare& cmp = Compare(),
                                  const Container& cont = Container())
           : comp(cmp), c(cont) {
              make_heap(c.begin(),c.end(),comp);
          }
          void push(const value_type& x) {
              c.push_back(x);
              push_heap(c.begin(),c.end(),comp);
          }
          void pop() {
              pop_heap(c.begin(),c.end(),comp);
              c.pop_back();
          }
          bool              empty() const { return c.empty(); }
          size_type         size() const  { return c.size(); }
          const value_type& top() const   { return c.front(); }
          …
        };
    }
```

These algorithms are described in Section 11.9.4, page 604.

Note that, unlike other container adapters, no comparison operators are defined. See Section 12.4, page 645, for details of the provided members and operations.