@toc

优先级队列

std::priority_queue

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

这里的Compare是一个仿函数/也叫函数对象

```
    less<int> ls;
    cout << ls(1, 2) << endl;
    greater<int> gt;
    cout<<gt(7,4)<<endl;
    //像less/greater就是仿函数，而ls/gt就是函数对象
```

---

priority_queue

STL实现的优先级队列是：**默认大的优先级高--给的仿函数是less，控制小的优先级高--给一个greater的仿函数(他是包含在functional的头文件中)**，它的底层实现就是堆(heap)

```
    priority_queue<int> pq;
    //priority_queue<int,vector<int>,greater<int>> pq;
    sort(pq.begin(), pq.end(),greater<int>());//从大到小
    //在上面的优先级队列中加的是greater<int> ---->因为那是一个模板参数，传的是类型
    //而sort是一个函数，里面传的是对象，这里用了一个匿名对象
    //如果数据类型不支持比较，或者比较的方式不是自己想要的那么就需要自己实现一个仿函数
```

(constructor)	Construct priority queue (public member function)
empty	Test whether container is empty (public member function)
size	Return size (public member function)
top	Access top element (public member function)
push	Insert element (public member function)
emplace C++11	Construct and insert element (public member function)
pop	Remove top element (public member function)
swap C++11	Swap contents (public member function)

具体可以去priority_queue - C++ Reference (cplusplus.com)查询

---

priority_queue的模拟实现

```
//优先级队列的底层实现其实就是一个堆(heap),他的模板参数用到了仿函数，默认情况下是大堆，也就是从大到小排列
//当输入greater的时候，会变成小堆排序，实现的话可以直接使用库里提供的push_heap和pop_heap，但是这里选择
//重新实现一下堆排的向上调整和向下调整
namespace yyr
{
    template<class T>
```

```
    struct less
    {
        bool operator()(const T& x, const T& y) const
        {
            return x < y;
        }
    };
    template<class T>
    struct greater
    {
        bool operator()(const T& x, const T& y) const
        {
            return x > y;
        }
    };
    //这里源码中给的默认容器是vector,Compare就是仿函数，默认是less
    template<class T,class Container = vector<T>,class Compare= less<T>>
    class priority_queue
    {
    private:
        void adjust_up(size_t child)
        {
            Compare com;
            size_t parent = (child - 1) / 2;
            while (child > 0)//如果用parent来当控制条件就必须是>=0,否则会少判断一次,而且
如果用parent>=0就不能使用size_t
            {
                //if (_con[parent] < _con[child])
                if(com(_con[parent],_con[child]))
                {
                    swap(_con[parent], _con[child]);
                    child = parent;
                    parent = (child - 1) / 2;
                }
                else
                {
                    break;
                }
            }
        }
        void adjust_down(size_t parent)
        {
            Compare com;
            size_t child = parent * 2 + 1;//左孩子
            while (child < _con.size())
            {
                //if (child + 1 < _con.size() && _con[child] < _con[child + 1])
                if (child + 1 < _con.size() && com(_con[child],_con[child + 1]))
                {
                    ++child;
                }

                //if (_con[parent] < _con[child])
                if(com(_con[parent],_con[child]))
                {
                    swap(_con[parent], _con[child]);
                    parent = child;
                    child = parent * 2 + 1;
```

```
                }
                else
                {
                    break;
                }
            }
        }
    public:
        priority_queue(const Container& val= Container())
            :_con(val)
        {}
        template<class InputIterator>
        priority_queue(InputIterator first,InputIterator last)
            :_con(first,last)//直接调用默认容器的迭代器构造
        {
            //然后开始实现建堆操作
            for (int i = (_con.size() - 1 - 1) / 2; i >= 0; --i)
            {
                adjust_down(i);
            }
        }
        void push(const T& val)
        {
            //因为是尾插元素，所以需要向上去调整
            _con.push_back(val);
            adjust_up(_con.size()-1);
        }
        void pop()
        {
            //队列肯定是从头开始出数据，但因为底层是堆，所以需要将头尾元素交换
            //然后删除尾部元素，同时进行向下调整
            swap(_con[0], _con[_con.size() - 1]);
            _con.pop_back();
            adjust_down(0);
        }
        const T& top() const
        {
            return _con[0];
        }
        bool empty() const
        {
            return _con.empty();
        }
    private:
        Container _con;
    };
}
```