优先队列 priority_queue 详解

一句话，优先队列，是一种可以自动排序的队列。

一、优先队列的头文件和声明

首先，你需要在程序首部加入

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

这两行。

using namespace std; 这句话，代表，使用一个叫做“std”的 namespace，namespace 里面封存了一系列东西，比方说奇异的数据结构和奇异的函数。当打开了 namespace 以后，就跟打开了头文件的本质是一样的，都是可以直接用它里面封存的函数。
不同之处在什么地方？就是不开 namespace 的使用，在你想用的（以 std 为例）函数面前加上“std::”即可。
例如，std::sort(a+1,a+1+N); 之类的。

其次，一个优先队列声明的基本格式是：
priority_queue<结构类型> 队列名;

比如：

```
priority_queue <int> i;
priority_queue <double> d;
```

不过，我们最为常用的是这几种：

```
priority_queue <node> q;
//node 是一个结构体
//结构体里重载了‘<’小于符号
priority_queue <int,vector<int>,greater<int> > q;
//不需要#include<vector>头文件
//注意后面两个“>”不要写在一起，“>>”是右移运算符
priority_queue <int,vector<int>,less<int> >q;
```

二、优先队列基本操作

以一个名为 q 的优先队列为例。

```
q.size();//返回 q 里元素个数
q.empty();//返回 q 是否为空，空则返回 1，否则返回 0
q.push(k);//在 q 的末尾插入 k
q.pop();//删掉 q 的第一个元素
```

q.top();//返回 q 的第一个元素

优先队列的特性
上文已经说过了，自动排序。
怎么个排法呢？
在这里介绍一下：

三、默认的优先队列（非结构体结构）

```
priority_queue <int> q;
```

这样的优先队列是怎样的？让我们写程序验证一下。

```
#include<cstdio>
#include<queue>
using namespace std;
priority_queue <int> q;
int main()
{
    q.push(10),q.push(8),q.push(12),q.push(14),q.push(6);
    while(!q.empty())
        printf("%d ",q.top()),q.pop();
}
```

程序大意就是在这个优先队列里依次插入 10、8、12、14、6，再输出。
结果是什么呢？
14 12 10 8 6
也就是说，它是按从大到小排序的！

四、默认的优先队列（结构体，重载小于）

先看看这个结构体是什么。

```
struct node
{
    int x,y;
    bool operator < (const node & a) const
    {
        return x<a.x;
    }
};
```

这个 node 结构体有两个成员，x 和 y，它的小于规则是 x 小者小。
再来看看验证程序：

```
#include<cstdio>
#include<queue>
using namespace std;
struct node
{
    int x,y;
```

```
    bool operator < (const node & a) const
    {
        return x<a.x;
    }
}k;
priority_queue <node> q;
int main()
{
    k.x=10,k.y=100; q.push(k);
    k.x=12,k.y=60; q.push(k);
    k.x=14,k.y=40; q.push(k);
    k.x=6,k.y=80; q.push(k);
    k.x=8,k.y=20; q.push(k);
    while(!q.empty())
    {
        node m=q.top(); q.pop();
        printf("(%d,%d) ",m.x,m.y);
    }
}
```

程序大意就是插入(10,100),(12,60),(14,40),(6,20),(8,20)这五个 node。
再来看看它的输出：
(14,40) (12,60) (10,100) (8,20) (6,80)

它也是按照重载后的小于规则，从大到小排序的。
好好看看这句话！（这是默认规则）

如果把小于的定义修改一下，如下：

```
struct node
{
    int x,y;
    bool operator < (const node & a) const
    {
        return x>a.x;
    }
};
```

则程序输出：
(6,80) (8,20) (10,100) (12,60) (14,40)
结果就是从小到大排序！

为何会这样，相反的逻辑？（追踪 STL 源码可以发现原因）

```
template<typename _Tp>
    struct greater : public binary_function<_Tp, _Tp, bool>
    {
      bool
      operator()(const _Tp& __x, const _Tp& __y) const
```

```
        { return __x > __y; }
      };
  template<typename _Tp>
      struct less : public binary_function<_Tp, _Tp, bool>
      {
        bool
        operator()(const _Tp& __x, const _Tp& __y) const
        { return __x < __y; }
      };
```

发现没有，原来
less 是从大到小，greater 是从小到大！

五、less 和 greater 优先队列

还是以 int 为例，先来声明：

```
priority_queue <int,vector<int>,less<int> > p;
priority_queue <int,vector<int>,greater<int> > q;
```

再次强调：“>”不要两个拼在一起。

话不多说，上程序和结果：

```
#include<cstdio>
#include<queue>
using namespace std;
priority_queue <int,vector<int>,less<int> > p;
priority_queue <int,vector<int>,greater<int> > q;
int a[5]={10,12,14,6,8};
int main()
{
    for(int i=0;i<5;i++)
        p.push(a[i]),q.push(a[i]);

    printf("less<int>:");
    while(!p.empty())
        printf("%d ",p.top()),p.pop();

    printf("\ngreater<int>:");
    while(!q.empty())
        printf("%d ",q.top()),q.pop();
}
```

结果：

```
less<int>:14 12 10 8 6 greater<int>:6 8 10 12 14
```

所以，我们可以知道，less 是从大到小，greater 是从小到大。

作个总结
为了方便，在平时，建议大家写：

```
priority_queue<int,vector<int>,less<int> >q;
priority_queue<int,vector<int>,greater<int> >q;
```

平时如果用从大到小，不用后面的 vector<int>,less<int>，可能到时候要改成从小到大，你反而会搞忘怎么写 greater<int>，反而得不偿失。

六、另一种排序方法

有可能遇到这种情况：不想用**重载小于**一个结构体的优先队列，要按照各种不一样的规则排序。

当然，如果不是优先队列而是数组，我们就会多写几个 bool 函数塞到 sort 里面来改变它的小于规则，比如：

```
struct node
{
    int fir,sec;
}arr[2030];

bool cmp1(node x,node y)
{
    return x.fir<y.fir;   //当一个 node x 的 fir 值小于另一个 node y 的 fir 值时，称 x<y
}

bool cmp2(node x,node y)
{
    return x.sec<y.sec;   //当一个 node x 的 sec 值小于另一个 node y 的 sec 值时，称 x<y
}

bool cmp3(node x,node y)
{
    return x.fir+x.sec<y.fir+y.sec;   //当一个 node x 的 fri 值和 sec 值的和小于另一个 node y 的 fir 值和 sec 值的和时，称 x<y
}

int main()
{
    scanf("%d",&n);
    for(int i=1;i<=n;i++) scanf("%d %d",&arr[i].fir,&arr[i].sec);

    puts("\n--------------------");
    sort(arr+1,arr+1+n,cmp1); for(int i=1;i<=n;i++) printf("%d. {%d %d}\n",i,arr[i].fir,arr[i].sec);
}

    puts("\n--------------------");
    sort(arr+1,arr+1+n,cmp2); for(int i=1;i<=n;i++) printf("%d. {%d %d}\n",i,arr[i].fir,arr[i].sec);
```

```
}

    puts("\n--------------------");
    sort(arr+1,arr+1+n,cmp3); for(int i=1;i<=n;i++) printf("%d. {%d %d}\n",i,arr[i].fir,arr[i].sec);
}
```

但是优先队列可没有 sort 那么灵活想用什么作小于规则用什么作小于规则，它只会用一个固定的小于规则。

所以如果想把一个队列按不同的方式优先，就要：

```
#include<queue>
#include<cstdio>
#include<cstring>
#include<iostream>
#include<algorithm>
using namespace std;

int n;
struct node
{
    int fir,sec;
    void Read() {scanf("%d %d",&fir,&sec);}
}input;

struct cmp1
{
    bool operator () (const node &x,const node &y) const
    {
        return x.fir<y.fir;
    }
};//当一个 node x 的 fir 值小于另一个 node y 的 fir 值时，称 x<y

struct cmp2
{
    bool operator () (const node &x,const node &y) const
    {
        return x.sec<y.sec;
    }
};//当一个 node x 的 sec 值小于另一个 node y 的 sec 值时，称 x<y

struct cmp3
{
    bool operator () (const node &x,const node &y) const
    {
        return x.fir+x.sec<y.fir+y.sec;
    }
};//当一个 node x 的 fri 值和 sec 值的和小于另一个 node y 的 fir 值和 sec 值的和时，称 x<y
```

```
priority_queue<node,vector<node>,cmp1> q1;
priority_queue<node,vector<node>,cmp2> q2;
priority_queue<node,vector<node>,cmp3> q3;

int main()
{
    scanf("%d",&n);
    for(int i=1;i<=n;i++) input.Read(),q1.push(input),q2.push(input),q3.push(input);

    printf("\ncmp1:\n");
    while(!q1.empty()) printf("(%d,%d) ",q1.top().fir,q1.top().sec),q1.pop();

    printf("\n\ncmp2:\n");
    while(!q2.empty()) printf("(%d,%d) ",q2.top().fir,q2.top().sec),q2.pop();

    printf("\n\ncmp3:\n");
    while(!q3.empty()) printf("(%d,%d) ",q3.top().fir,q3.top().sec),q3.pop();
}
```

读入：

```
7
1 2
2 1
6 9
9 6
-100 100
-500 20
4000 -3000
```

输出：

```
cmp1:
(4000,-3000) (9,6) (6,9) (2,1) (1,2) (-100,100) (-500,20)

cmp2:
(-100,100) (-500,20) (6,9) (9,6) (1,2) (2,1) (4000,-3000)

cmp3:
(4000,-3000) (6,9) (9,6) (1,2) (2,1) (-100,100) (-500,20)
```

我们可以发现啊，priority_queue <int,vector<int>,less<int> > p;的那个 less<int>其实就代表这个优先队列的小于规则，所以把这个换成 cmp1 就会有上述效果，所以说，一定要记得写全称！

记住，优先队列，是一种可以自动排序的队列！

最完整的声明形如：
priority_queue< 结构名, vector<结构名> , greater/less<结构名> > 队列名;

可以简写为 priority_queue<结构名>,不过这样只能从大到小了。
三个结构名请保持一致，如 int,double,long long，包括结构体（struct 等）。
要求是这个结构要有小于的规则——你要告诉它怎么比较大小，它才能帮你排序。
系统自带的数据结构的小于规则是显然的，对于结构体，需要通过重载运算符等方式规定，如：

```
struct point
{
    int x,y;
    bool operator < (const point &p) const
    {
        return x*x+y*y<p.x*p.x+p.y*p.y;
        //假设这是平面上的两个点，规定两个点的大小关系为距离原点的距离小者小。
        //这个函数的意思是，当你使用小于运算符判断一个点（假设是 a）与另一个点
        //（函数里的 p）的大小关系时，系统会判断 a.x*a.x+a.y*a.y 是否<p.x*p.x+p.y*p.y
        //如果上述式子成立，就说明 a 点是小于 p 点的（return 1 ; ->小于运算符得出的结果为真)
    }
};
```

```
priority_queue<point> QP;
```
greater 代表升序，即从小到大，
less 代表降序，即从大到小，与缩减版无异。
如果不想用重载小于，就新建一个结构体并重载等号，在第三项里填入这个结构体的名字。

然后是各类操作
```
q.size();//返回 q 里元素个数
q.empty();//返回 q 是否为空，空则返回 1，否则返回 0
q.push(k);//在 q 的末尾插入 k
q.pop();//删掉 q 的第一个元素
q.top();//返回 q 的第一个元素
```