# **队列容器queue**

**头文件：** #include < queue >  
**内部数据结构：** 连续或分段连续存储数组（两端  
开口的数组）  
**特点：** 获取元素效率较高，插入和删除的效率较高  
**注意事项：** 禁止在首尾之外的地方访问/插入元素

### **构造函数**

| **用法** | **解释** |
| --- | --- |
| queue < Type > q; | 新建Type类型的queue |
| queue < Type > q2(q1); | q2是q1的副本 |
| queue < Type > q2=q1; | 赋值，两者大小可以不一样 |

注意：圆括号()只能在定义时使用

### **修改数据**

不能通过下标的形式插入新数据，可以通过下标修改已有数据，但易访问越界出错

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| q.push(a); | 给队尾插入数据 | O(1) |
| q.emplace(a); | 给队尾插入数据 | O(1) |
| q1.swap(v2); | 交换q1和q2 | O(1) |

emplace效率比push高一点，因为不触发拷贝构造和转移构造

### **访问元素**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| q.front(); | 返回首元素 | O(1) |
| q.back(); | 返回最后一个元素 | O(1) |

### **删除元素**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| q.pop(); | 删除队首元素 | O(1) |

### **长度&容量**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| q.size(); | 获取q的长度 | O(1) |
| q.empty(); | 为空返回true，不为空返回false | O(1) |

# **队列容器deque**

**头文件：** #include < deque >  
**内部数据结构：** 连续或分段连续存储数组（两端  
开口的数组）  
**特点：** 获取元素效率较高，插入和删除的效率较高  
**注意事项：** 禁止在首尾之外的地方访问/插入元素

### **构造函数**

| **用法** | **解释** |
| --- | --- |
| deque < Type > deq; | 新建Type类型的deque |
| deque < Type > deq2(deq1); | deq2是deq1的副本 |
| deque < Type > deq2=deq1; | 赋值，两者大小可以不一样 |
| deque < Type > deq(n); | deq包含n个元素 |
| deque < Type > deq(n,i); | deq包含n个元素，全部初始化为i |
| deque < Type > deq{a,b,c,...}; | 包含初始个数的的元素，每个元素赋予相应初值 |
| deque < Type > deq={a,b,c,...}; | 同上 |
| deque<deque< Type > > deq; | 嵌套deque，相当于二维数组 |

注意：圆括号()只能在定义时使用

### **修改数据**

不能通过下标的形式插入新数据，可以通过下标修改已有数据，但易访问越界出错

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| deq.push\_front(a); | 给队首插入数据 | O(1) |
| deq.push\_back(a); | 给队尾插入数据 | O(1) |
| deq.insert(it, val); | 向迭代器it指向的元素前插入新元素deqal | O(N) |
| deq.insert(it, n, x); | 向迭代器it指向的元素前插入n个x | O(N) |
| deq.insert(it, first, last); | 将迭代器指定的序列[first, last)插入到迭代器it指向的元素前 | O(N) |
| deq.emplace(it,a); | 从it位置开始插入元素a | O(N) |
| deq.emplace\_front(a); | 给deq前插入元素a | O(1) |
| deq.emplace\_back(a); | 给deq后插入元素a | O(1) |
| deq.assign(deq2.begin(),deq2.end()); | deq重新初始化为deq2的值 | O(N) |
| deq.assign(n,val); | 重新初始化n个数，值为val | O(N) |
| deq.assign({a1,a2,a3,...}); | 重新初始化 | O(N) |
| deq1.swap(deq2); | 交换deq1和deq2 | O(1) |

emplace效率比insert和push\_back高一点，因为不触发拷贝构造和转移构造

### **访问元素**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| deq[i]; | 访问第i个元素 | O(1) |
| [deq.at](http://deq.at/)(i); | 访问第i个元素，会检查边界是否越界 | O(1) |
| deq.front(); | 返回首元素 | O(1) |
| deq.back(); | 返回最后一个元素 | O(1) |
| deq.begin(); | 返回首元素的**地址** | O(1) |
| deq.end(); | 返回最后一个元素**后一位的地址** | O(1) |
| deq.rbegin(); | 返回最后一个元素**地址**(++取前一个元素) | O(1) |
| deq.rend(); | 返回第一个元素之前的**虚拟地址**(--取后一个元素) | O(1) |
| cbegin(),cend(),crbegin(),crend() | 返回一个const迭代器 | O(1) |

### **删除元素**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| deq.pop\_front(); | 删除队首元素 | O(1) |
| deq.pop\_back(); | 删除队尾元素 | O(1) |
| deq.erase(it); | 删除迭代器it指向的元素 | O(N) |
| deq.erase(first,last); | 删除迭代器指定区间 [first,last) 的元素 | O(N) |
| deq.clear(); | 清空全部元素 | O(N) |

### **长度&容量**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| deq.size(); | 获取deq的使用长度 | O(1) |
| deq.max\_size(); | 返回在当前平台下能给Type类型的deque分配的最大容量 | O(1) |
| deq.resize(n,val); | 修改deq的使用长度为n，并初始化为val(val可略去) | O(N) |
| deq.empty(); | 为空返回true，不为空返回false | O(1) |

### **迭代器**

| **用法** | **解释** |
| --- | --- |
| deque:: iterator it; | 定义一个Type类型的迭代器it |
| deque::reverse\_iterator it; | 定义一个Type类型的反向迭代器it |
| for (auto it=deq.begin();it!=deq.end();it++) cout<<\*it<<endl; | 用auto遍历 |
| for (auto it: deq) cout<<it<<endl; | 用auto遍历 |

| **支持的操作** | **解释** |
| --- | --- |
| operator\* | 取内容 |
| operator-> | 取地址 |
| operator++ | 自增 |
| operator-- | 自减 |
| operator+ | 位置加 |
| operator- | 位置减 |
| operator+= | 加等于 |
| operator-= | 减等于 |

# **队列容器priority\_queue**

**头文件：** #include < priority\_queue >  
**内部数据结构：** 连续或分段连续存储数组（两端  
开口的数组）  
**特点：** 获取元素效率较高，插入和删除的效率较高  
**注意事项：** 禁止在首尾之外的地方访问/插入元素

### **构造函数**

| **用法** | **解释** |
| --- | --- |
| priority\_queue < Type > priq; | 新建Type类型的priority\_queue |
| priority\_queue < Type > priq2(priq1); | priq2是priq1的副本 |
| priority\_queue < Type > priq2=priq1; | 赋值，两者大小可以不一样 |
| priority\_queue<int,vector,greater> priq; | 从大到小排序，最小值优先 |
| priority\_queue<int,vector,less > priq; | 从小到大排序，最大值优先，后两个参数不写默认为这个 |

注意：圆括号()只能在定义时使用  
由于元素是从顶部弹出的，所以输出大小顺序是反的

#### **自定义优先级**

* 从小到大输出

priority\_queue<int, vector<int>, greater<int> > q;

* 从小到大输出

struct cmp{

bool operator ()(int x, int y)

{

return x > y;

}};

priority\_queue<int, vector<int>, cmp> q;

* 按照结构体的x从小到大输出

struct node{

int x, y;

friend bool operator < (node a, node b)

{

return a.x > b.x;

}}no[macn];

priority\_queue<node> q;

### **修改数据**

不能通过下标的形式插入新数据，可以通过下标修改已有数据，但易访问越界出错

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| priq.push(a); | 给队尾插入数据 | O(1) |
| priq.emplace(a); | 给队尾插入数据 | O(1) |
| priq1.swap(v2); | 交换priq1和priq2 | O(1) |

emplace效率比push高一点，因为不触发拷贝构造和转移构造

### **访问元素**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| priq.top(); | 返回队尾元素 | O(1) |

### **删除元素**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| priq.pop(); | 删除队尾元素 | O(1) |

### **长度&容量**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| priq.size(); | 获取priq的长度 | O(1) |
| priq.empty(); | 为空返回true，不为空返回false | O(1) |

# **列表容器list**

**头文件：** #include < list >  
**内部数据结构：** 双向环状链表  
**特点：** 获取元素效率很低，插入和删除的效率很高  
**注意事项：** 插入，迭代器不会失效。删除，指向被删除节点迭代器失效。

### **构造函数**

| **用法** | **解释** |
| --- | --- |
| list < Type > l; | 新建Type类型的list |
| list < Type > l2(l1); | l2是l1的副本 |
| list < Type > 12 = 11; | 赋值，两者大小可以不一样 |
| list < Type > 12 (l1.begin(),l1.end()); | 同上 |
| list < Type > l(n); | l包含n个元素 |
| list < Type > l(n, i); | l包含n个元素，全部初始化为i |
| list < Type > l{ a,b,c,... }; | 包含初始个数的的元素，每个元素赋予相应初值 |
| list < Type > l = { a,b,c,... }; | 同上 |

注意：圆括号()只能在定义时使用

### **修改数据**

不能通过下标的形式插入新数据，可以通过下标修改已有数据，但易访问越界出错

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| l.push\_back(a); l.push\_front(a); | 给表尾插入数据  给表首插入数据 | O(1) |
| l.insert(it, val); | 向迭代器it指向的元素前插入新元素val | O(N) |
| l.insert(it, n, x); | 向迭代器it指向的元素前插入n个x | O(N) |
| l.insert(it, first, last); | 将迭代器指定的序列[first, last)插入到迭代器it指向的元素前 | O(N) |
| l.emplace(it, a); | 在it位置插入a | O(1) |
| l.emplace\_back(a); l.emplace\_front(a); | 给l后面插入元素a 给l前面插入元素a | O(1) |
| l.assign(l2.begin(), l2.end()); | l重新初始化为l2的值 | O(N) |
| l.assign(n, i); | l包含n个元素，全部初始化为i | O(N) |
| l.assign({ a1,a2,a3,... }); | 重新初始化 | O(N) |
| l1.swap(l2); | 交换l1和l2 | O(1) |
| l1.merge(l2); | 合并两个有序链表到l1中 | O(N) |
| l1.splice(it,l2); | 在l1的it前插入l2所有元素 | O(1) |
| l1.splice(it1,l2,it2); | l1的it1位置插入l2的it2指向的元素 | O(1) |
| l.splice(l1.begin(),it1,it2,l1.end()); | l1的【begin到it2】和【it2到end】交换顺序 | O(N) |

emplace效率比insert和push\_back高一点，因为不触发拷贝构造和转移构造

### **访问元素**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| l.front(); | 返回首元素 | O(1) |
| l.back(); | 返回最后一个元素 | O(1) |
| l.begin(); | 返回首元素的 **地址** | O(1) |
| l.end(); | 返回最后一个元素 **后一位的地址** | O(1) |
| l.rbegin(); | 返回最后一个元素 **地址**(++取前一个元素) | O(1) |
| l.rend(); | 返回第一个元素之前的 **虚拟地址** (--取后一个元素) | O(1) |
| cbegin(), cend(), crbegin(), crend() | 返回一个const迭代器 | O(1) |
| l.sort(cmp); | 按照cmp排序 | O(N) |

sort的用法：

bool cmp(int a,int b)//从大到小排序{

return a > b;}int main(){

list<int> l{1,2,3,4};

l.sort(cmp);}

### **删除元素**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| l.pop\_back(); l.pop\_front(); | 弹出表尾数据  弹出表首数据 | O(1) |
| l.erase(it); | 删除迭代器it指向的元素 | O(N) |
| l.erase(first, last); | 删除迭代器指定区间[first, last) 的元素 | O(N) |
| l.remove(a); | “删除”元素a，其实是放到链表尾部，不减少size | O(N) |
| l.remove\_if(begin, end, op) | 移除区间[begin,end)中每一个“令op(elem)获得true”的元素 | O(N) |
| l.clear(); | 清空全部元素 | O(N) |
| l.unique(); | 去重，必须对有序链表操作 | O(N) |

### **长度&容量**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| l.size(); | 获取l的使用长度 | O(1) |
| l.max\_size(); | 返回在当前平台下能给Type类型的list分配的最大容量 | O(1) |
| l.resize(n, val); | 修改l的使用长度为n，并初始化为val(val可略去) | O(N) |
| l.empty(); | 为空返回true，不为空返回false | O(1) |

### **迭代器**

| **用法** | **解释** |
| --- | --- |
| list::iterator it; | 定义一个Type类型的迭代器it |
| list::reverse\_iterator it; | 定义一个Type类型的反向迭代器it |
| for (auto it = vec.begin(); it != vec.end(); it++)  cout << \*it << endl; | 用auto遍历 |
| for (auto it : vec)  cout << it << endl; | 用auto遍历 |

| **支持的操作** | **解释** |
| --- | --- |
| operator\* | 取内容 |
| operator-> | 取地址 |
| operator++ | 自增 |
| operator-- | 自减 |
| operator+ | 位置加 |
| operator- | 位置减 |
| operator+= | 加等于 |
| operator-= | 减等于 |

# **映射map**

**头文件：** #include < map >  
**内部数据结构：** 红黑树  
**特点：**

1. 键和值分开（模版有两个参数，前面是键后面是值）
2. 键唯一
3. 元素默认按键的升序排列

**注意事项：** 插入，迭代器不会失效。删除，指向被删除节点迭代器失效。

# **多重映射multimap**

**头文件：** #include < map >  
**内部数据结构：** 红黑树  
**特点：**

1. 键和值分开
2. 键可以不唯一
3. 元素默认按键的升序排列

**注意事项：** 插入，迭代器不会失效。删除，指向被删除节点迭代器失效。

### **构造函数**

| **用法** | **解释** |
| --- | --- |
| map < Type1,Type2 > s; | 新建键为Type1，值为Type2的map |
| map< Type1,Type2 > s2(s1); | s2是s1的副本 |
| map < Type1,Type2 > s2(s1.begin(), s1.end()); | 同上 |
| map < Type1,Type2 > s2 = s1; | 赋值，两者大小不一样也可以 |

注意：圆括号()只能在定义时使用

按value排序

struct cmp {

bool operator()(const pair<int,int>& lhs, const pair<int,int>& rhs)const

{

return lhs.second < rhs.second;

}};sort(mp.begin(),mp.end(),cmp());

**自定义键值的构造方法**  
方法一

struct Person {

string name;

int age;}p;struct cmp{

bool operator ()(const Person& p1,const Person& p2) const //注意这里的两个const必须要有

{

return (p1.age < p2.age) || (p1.age == p2.age && p1.name.length() < p2.name.length());

}};

map<Person, int, cmp> mp;

p.age = 1, p.name = "Jing";

mp[p] = 1;

cout << mp[p] << endl;

方法二

class Person {public:

string name;

int age;

Person(string n, int a) {

name = n;

age = a;

}

bool operator<(const Person& p) const //注意这里的两个const必须要有

{

return (age < p.age) || (age == p.age && name.length() < p.name.length());

}};int main() {

map<Person, int> mp;

mp[Person("Jing", 20)] = 1;

mp[Person("Wei", 21)] = 2;

for (auto i : mp)

{

cout << i.first.name << " " << i.first.age << " ";

cout << i.second << endl;

}

return 0;}

### **修改数据**

不能通过下标的形式插入新数据，可以通过下标修改已有数据，但易访问越界出错

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| mp.insert(make\_pair(val1,val2)); | 插入键为val1，值为val2的元素 | O(logN) |
| mp.insert(it, make\_pair(val1,val2)); | 向迭代器it指向位置插入数据 | O(1)~O(logN) |
| mp2.insert(mp1.begin(),mp2.end()); | 给mp2插入mp1[firsr,last)的数据 | O(NlogN) |
| mp.emplace(val1,val2); | 插入键为val1，值为val2的元素 | O(logN) |
| mp.emplace\_hint(it, val1, val2); | 向迭代器it指向位置插入数据 | O(logN) |
| mp1.swap(mp2); | 交换mp1和mp2 | O(1) |

emplace效率比insert高一点，因为不触发拷贝构造和转移构造

### **访问元素**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| mp[key]; | 访问键值key对应的元素 | O(logN) |
| [mp.at](http://mp.at/)(key); | 访问键值key对应的元素，会检查边界是否越界 | O(logN) |
| mp.begin(); | 返回首元素的 **地址** | O(1) |
| mp.end(); | 返回最后一个元素 **后一位的地址** | O(1) |
| mp.rbegin(); | 返回最后一个元素 **地址**(++取前一个元素) | O(1) |
| mp.rend(); | 返回第一个元素之前的 **虚拟地址** (--取后一个元素) | O(1) |
| cbegin(), cend(), crbegin(), crend() | 返回一个const迭代器 | O(1) |
| mp.find(val1); | 返回键val1的 **地址** | O(logN) |
| s.lower\_bound(key); | 返回大于等于key的第一个数的地址 | O(logN) |
| s.upper\_bound(key); | 返回大于键val1的第一个数的地址 | O(logN) |
| s.equal\_range(key); | 返回一对pair地址，第一个是lower\_bound后的地址，第二个是upper\_bound后的地址 | O(logN) |
| s.key\_comp(); | 比较键，小于返回1，否则返回0 | O(1) |
| s.value\_comp(); | 比较值，小于返回1，否则返回0 | O(1) |
| s.count(key); | 数s中有几个key，map中只有0和1 | O(1) |

multimap没有[]和at的功能，因为它的键值可以重复

**key\_comp()的用法**

map<string, int> mp, mp2;

mp.emplace("a", 20);

mp.emplace("b", 10);

map<string,int>::iterator it = mp.begin();

map<string,int>::reverse\_iterator it2 = mp.rbegin();

map<string, int>::key\_compare cmp = mp.key\_comp();

cout << (\*it).first << " " << (\*it2).first << endl;

cout << cmp((\*it).first, (\*it2).first) << endl;//begin的键肯定比end的键小，输出1//value\_comp这里改成cmp(\*it,\*it2)，其他同理//（但是这里测试后发现好像有点问题，value\_comp也是按key比较的,功能和key\_value一模一样）

**equal\_range(val)的用法**

pair<map<int>::iterator, map<int>::iterator> it;

it = s1.equal\_range(3);

cout << "the lower bound points to: " << \*it.first << '\n';

cout << "the upper bound points to: " << \*it.second << '\n';

### **删除元素**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| mp.erase(key); | 删除key对应元素 | O(logN) |
| s.erase(it); | 删除迭代器it指向的元素 | O(1) |
| s.erase(first, last); | 删除[first, last)区间的元素 | O(N) |
| s.clear(); | 清空全部元素 | O(N) |

### **长度&容量**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| s.size(); | 获取s的使用容量 | O(1) |
| s.max\_size(); | 返回在当前平台下能给Type类型的map分配的最大容量 | O(1) |
| s.empty(); | 为空返回true，不为空返回false | O(1) |

### **迭代器**

| **用法** | **解释** |
| --- | --- |
| map< Type1,Type2 >::iterator it; | 定义一个Type类型的迭代器it |
| map< Type1,Type2 >::reverse\_iterator it; | 定义一个Type类型的反向迭代器it |
| for (auto it = vec.begin(); it != vec.end(); it++)  cout << \*it << endl; | 用auto遍历 |
| for (auto it : vec)  cout << it << endl; | 用auto遍历 |

| **支持的操作** | **解释** |
| --- | --- |
| operator\* | 取内容 |
| operator-> | 取地址 |
| operator++ | 自增 |
| operator-- | 自减 |
| operator+ | 位置加 |
| operator- | 位置减 |
| operator+= | 加等于 |
| operator-= | 减等于 |

# **set与multiset**

## **集合set**

**头文件：** #include < set >  
**内部数据结构：** 红黑树（平衡检索二叉树）  
**特点：**

1. 键（关键字）和值（数据）相等
2. 键唯一，无重复元素
3. 元素默认按升序排列
4. 插入删除效率高

**注意事项：** 插入，迭代器不会失效。删除，指向被删除节点迭代器失效。

## **多重集合multiset**

**头文件：** #include < set >  
**内部数据结构：** 红黑树（平衡检索二叉树）  
**特点：**

1. 键和值相等
2. 键可以不唯一
3. 元素默认按升序排列

**注意事项：** 插入，迭代器不会失效。删除，指向被删除节点迭代器失效。

**set与multiset的函数和用法一样，区别只有multiset可以插入重复数据**

### **构造函数**

| **用法** | **解释** |
| --- | --- |
| set < Type > s; | 新建Type类型的set |
| set < Type > s2(s1); | s2是s1的副本 |
| set < Type > s2 = s1; | 赋值,两者大小不一样也可以 |
| set < Type > s2(s1.begin(),s1.end()); | 同上 |
| int a[5]={1,2,3,4,5}; set < int > s(a,a+5); | 包含初始个数的的元素，每个元素赋予相应初值 |

注意：圆括号()只能在定义时使用

**自定义类型的构造方法**

struct S {

int x, y;};struct cmp {

bool operator()(const S& a, const S& b) {

return a.x < b.x || a.x == b.x && a.y < b.y;

}};

multiset<S, cmp> a;

### **修改数据**

不能通过下标的形式插入新数据，可以通过下标修改已有数据，但易访问越界出错

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| s.insert(val); | 插入元素val | O(logN) |
| s.insert(it, val); | 向迭代器it指向位置插入val | O(1)~O(logN) |
| int a[3]={1,2,3}; s.insert(a,a+3); | 将序列[first, last)插入到set中 | O(NlogN) |
| s.emplace(val); | 插入元素val | O(logN) |
| s.emplace\_hint(it,val); | 向迭代器it指向位置val | O(1)~O(logN) |
| s1.swap(s2); | 交换s1和s2 | O(1) |

emplace效率比insert高一点，因为不触发拷贝构造和转移构造

### **访问元素**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| s.begin(); | 返回首元素的 **地址** | O(1) |
| s.end(); | 返回最后一个元素 **后一位的地址** | O(1) |
| s.rbegin(); | 返回最后一个元素 **地址**(++取前一个元素) | O(1) |
| s.rend(); | 返回第一个元素之前的 **虚拟地址** (--取后一个元素) | O(1) |
| cbegin(), cend(), crbegin(), crend() | 返回一个const迭代器 | O(1) |
| s.find(val); | 返回val的**地址** | O(logN) |
| s.lower\_bound(val); | 返回大于等于val的第一个数的地址 | O(logN) |
| s.upper\_bound(val); | 返回大于val的第一个数的地址 | O(logN) |
| s.equal\_range(val); | 返回一对pair地址，第一个是lower\_bound后的地址，第二个是upper\_bound后的地址 | O(logN) |
| s.key\_comp(); | 比较键，小于返回1，否则返回0 | O(1) |
| s.value\_comp(); | 比较值，小于返回1，否则返回0 | O(1) |
| s.count(val); | 数s中有几个val，set中只有0和1 | O(1) |

**key\_comp()的用法**

int a[5] = { 1,3,4,5,6 };

set<int> s1(a, a + 5);

set<int>::iterator it = s1.begin();

set<int>::reverse\_iterator it2 = s1.rbegin();

set<int>::key\_compare cmp = s1.key\_comp();

cout << \*it << " " << \*it2 << endl;

cout << cmp(\*it, \*it2) << endl;//value\_comp同理

**equal\_range(val)的用法**

pair<set<int>::iterator, set<int>::iterator> it;

it = s1.equal\_range(3);

cout << "the lower bound points to: " << \*it.first << '\n';

cout << "the upper bound points to: " << \*it.second << '\n';

### **删除元素**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| s.erase(val); | 删除元素val | O(logN) |
| s.erase(it); | 删除迭代器it指向的元素 | O(1) |
| s.erase(first,last); | 删除[first,last)区间的元素 | O(N) |
| s.clear(); | 清空全部元素 | O(N) |

### **长度&容量**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| s.size(); | 获取s的使用容量 | O(1) |
| s.max\_size(); | 返回在当前平台下能给Type类型的set分配的最大容量 | O(1) |
| s.empty(); | 为空返回true，不为空返回false | O(1) |

### **迭代器**

| **用法** | **解释** |
| --- | --- |
| set::iterator it; | 定义一个Type类型的迭代器it |
| set::reverse\_iterator it; | 定义一个Type类型的反向迭代器it |
| for (auto it = vec.begin(); it != vec.end(); it++) cout << \*it << endl; | 用auto遍历 |
| for (auto it : vec) cout << it << endl; | 用auto遍历 |

| **支持的操作** | **解释** |
| --- | --- |
| operator\* | 取内容 |
| operator-> | 取地址 |
| operator++ | 自增 |
| operator-- | 自减 |
| operator+ | 位置加 |
| operator- | 位置减 |
| operator+= | 加等于 |
| operator-= | 减等于 |

# **栈容器stack**

**头文件：** #include < stack >  
**内部数据结构：** 连续存储的数组形式（一端开口）  
**特点：** 获取元素效率较高，插入和删除的效率较高  
**注意事项：** 禁止在顶部之外的地方访问/插入元素

### **构造函数**

| **用法** | **解释** |
| --- | --- |
| stack < Type > sta; | 新建Type类型的stack |
| stack < Type > sta2(sta1); | sta2是sta1的副本 |
| stack < Type > sta2 = sta1; | 赋值，两者大小可以不一样 |

注意：圆括号()只能在定义时使用

### **修改数据**

不能通过下标的形式插入新数据，可以通过下标修改已有数据，但易访问越界出错

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| sta.push(a); | 给栈顶插入数据 | O(1) |
| sta.emplace(a); | 给栈顶插入数据 | O(1) |
| sta1.swap(sta2); | 交换sta1和sta2 | O(1) |

emplace效率比push高一点，因为不触发拷贝构造和转移构造

### **访问元素**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| sta.top(); | 返栈顶元素 | O(1) |

### **删除元素**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| sta.pop(); | 删除栈顶元素 | O(1) |

### **长度&容量**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| sta.size(); | 获取sta的长度 | O(1) |
| sta.empty(); | 为空返回true，不为空返回false | O(1) |

# **向量容器vector**

**头文件：** #include < vector >  
**内部数据结构：** 连续存储的数组形式（一端开口的组）  
**特点：** 高效率获取元素，插入和删除效率很低  
**注意事项：** 插入和删除会使迭代器失效

### **构造函数**

| **用法** | **解释** |
| --- | --- |
| vector < Type > v; | 新建Type类型的vector |
| vector < Type > v2(v1); | v2是v1的副本 |
| vector < Type > v2=v1; | 赋值，两者大小可以不一样 |
| vector < Type > v(n); | v包含n个元素 |
| vector < Type > v(n,i); | v包含n个元素，全部初始化为i |
| vector < Type > v{a,b,c,...}; | 包含初始个数的的元素，每个元素赋予相应初值 |
| vector < Type > v={a,b,c,...}; | 同上 |
| vector<vector< Type > > v; | 嵌套vector，相当于二维数组 |

注意：圆括号()只能在定义时使用

### **修改数据**

不能通过下标的形式插入新数据，可以通过下标修改已有数据，但易访问越界出错

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| v.push\_back(a); | 给表尾插入数据 | O(1) |
| v.insert(it, val); | 向迭代器it指向的元素前插入新元素val | O(N) |
| v.insert(it, n, x); | 向迭代器it指向的元素前插入n个x | O(N) |
| v.insert(it, first, last); | 将迭代器指定的序列[first, last)插入到迭代器it指向的元素前 | O(N) |
| v.emplace(it,a1,a2,a3,...); | 从it位置开始插入a1,a2,a3,... | O(N) |
| v.emplace\_back(a); | 给v后面插入元素a | O(1) |
| v.assign(v2.begin(),v2.end()); | v重新初始化为v2的值 | O(N) |
| v.assign({a1,a2,a3,...}); | 重新初始化 | O(N) |
| v1.swap(v2); | 交换v1和v2（其实是交换了首尾指针） | O(1) |

emplace效率比insert和push\_back高一点，因为不触发拷贝构造和转移构造

### **访问元素**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| v[i]; | 访问第i个元素 | O(1) |
| [v.at](http://v.at/)(i); | 访问第i个元素，会检查边界是否越界 | O(1) |
| v.front(); | 返回首元素 | O(1) |
| v.back(); | 返回最后一个元素 | O(1) |
| v.data(); | 返回一个指针，指向vector存储数据的地址 | O(1) |
| v.begin(); | 返回首元素的**地址** | O(1) |
| v.end(); | 返回最后一个元素**后一位的地址** | O(1) |
| v.rbegin(); | 返回最后一个元素**地址**(++取前一个元素) | O(1) |
| v.rend(); | 返回第一个元素之前的**虚拟地址**(--取后一个元素) | O(1) |
| cbegin(),cend(),crbegin(),crend() | 返回一个const迭代器 | O(1) |

### **删除元素**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| v.pop\_back(); | 删除尾端元素 | O(1) |
| v.erase(it); | 删除迭代器it指向的元素 | O(N) |
| v.erase(first,last); | 删除迭代器指定区间 [first,last) 的元素 | O(N) |
| v.clear(); | 清空全部元素 | O(N) |

### **长度&容量**

| **用法** | **解释** | **时间复杂度** |
| --- | --- | --- |
| v.size(); | 获取v的使用长度 | O(1) |
| v.max\_size(); | 返回在当前平台下能给Type类型的vector分配的最大容量 | O(1) |
| v.resize(n,val); | 修改v的使用长度为n，并初始化为val(val可略去) | O(N) |
| v.capacity(); | 返回vector的当前容量 | O(1) |
| v.empty(); | 为空返回true，不为空返回false | O(1) |
| v.reserve(n); | 添加一段容量n | O(N) |

### **迭代器**

| **用法** | **解释** |
| --- | --- |
| vector:: iterator it; | 定义一个Type类型的迭代器it |
| vector::reverse\_iterator it; | 定义一个Type类型的反向迭代器it |
| for (auto it=vec.begin();it!=vec.end();it++) cout<<\*it<<endl; | 用auto遍历 |
| for (auto it: vec) cout<<it<<endl; | 用auto遍历 |

| **支持的操作** | **解释** |
| --- | --- |
| operator\* | 取内容 |
| operator-> | 取地址 |
| operator++ | 自增 |
| operator-- | 自减 |
| operator+ | 位置加 |
| operator- | 位置减 |
| operator+= | 加等于 |
| operator-= | 减等于 |