STL 容器函数快速入门

何南宁 *

2024 年 12 月 7 日

*本文基于 C++20

目录

1 vector 动态数组

```
vector<element_type> v; 动态数组

v.size()
返回动态数组 v 的元素的个数 (返回类型为 usinged) O(1)
vector<int> v1 = {1, 2, 3};
cout << v1.size() << endl; // 3
vector<int> v2 = {1, 2, 3, 4};
cout << v2.size() << endl; // 4

v.empty()
判断动态数组 v 是否为空
空则返回 true, 非空则返回 false(返回类型是 bool) O(1)
vector<int> v1 = {1, 2, 3};
cout << v1.empty() << endl; // 0
vector<int> v2 = {};
cout << v2.empty() << endl; // 1

v.front()
返回动态数组 v 的第一个元素 O(1)
vector<int> v1 = {1, 2, 3};
cout << v1.front() << endl; // 1
vector<int> v2 = {4, 3, 2, 1};
cout << v2.front() << endl; // 4

v.back()
返回动态数组 v 的最后一个元素 O(1)
vector<int> v1 = {1, 2, 3};
cout << v1.back() << endl; // 3
vector<int> v2 = {4, 3, 2, 1};
cout << v2.back() << endl; // 1

v.begin()
返回指向动态数组 v 第一个元素的迭代器 (正向) O(1)
vector<int> v1 = {1, 2, 3};
cout << *v1.begin() << endl; // 1
```

```
vector<int> v2 = {4, 3, 2, 1};
cout << *v2.begin() << endl; // 4
```

v.end()
返回指向动态数组 v 的最后一个元素的后一个位置的迭代器（正向）O(1)

v.rbegin()
返回指向动态数组 v 的最后一个元素的迭代器（逆向）O(1)

```
vector<int> v1 = {1, 2, 3};
cout << *v1.rbegin() << endl; // 3
vector<int> v2 = {4, 3, 2, 1};
cout << *v2.rbegin() << endl; // 1
```

v.rend()
返回指向动态数组 v 的第一个元素的前一个位置的迭代器（逆向）O(1)

v.push_back(element)
添加一个元素 element 在动态数组 v 的最后面 O(1)

```
vector<int> v;
v.push_back(1), v.push_back(2), v.push_back(3);
for(int i = 0; i < v.size(); i++){
    cout << v[i] << " ";
}
//1 2 3
```

v.emplace_back(element)
添加一个元素 element 在动态数组 v 的后面 O(1)

```
vector<int> v;
v.emplace_back(1), v.emplace_back(2), v.emplace_back(3);
for(int i = 0; i < v.size(); i++){
    cout << v[i] << " ";
}
//1 2 3
```

v.pop_back()
删除动态数组 v 的最后一个元素 O(1)

```
vector<int> v = {1, 2, 3};
```

```
v.pop_back();
for(int i = 0; i < v.size(); i++){
    cout << v[i] << " ";
}
//1 2
```

v.clear()
清除动态数组 v 的所有元素 O(n)

```
vector<int> v = {1, 2, 3};
v.clear();
cout << v.size(); // 0
//v = {}
```

v.erase(it)
删除迭代器 it 指向的元素 O(n)

```
vector<int> v = {1, 2, 3};
v.erase(v.begin() + 1);
for(int i = 0; i < v.size(); i++){
    cout << v[i] << " ";
}
//1 3
```

v.erase(first, last)
删除迭代器 [first, last) 之间的所有元素 O(n)

```
vector<int> v = {1, 2, 3, 4};
v.erase(v.begin() + 1, v.begin() + 3);
for(int i = 0; i < v.size(); i++){
    cout << v[i] << " ";
}
//1 4
```

v.insert(it, x)
在迭代器 it 处插入一个元素 x O(n)

```
vector<int> v = {1, 2, 3};
v.insert(v.begin() + 1, 4);
for(int i = 0; i < v.size(); i++){
    cout << v[i] << " ";
```

```
}
//1 4 2 3


v.resize(n, v) (
修改动态数组的长度为 n 并且赋值为 v(默认为 0) O(n)
vector<int> v1 = {1, 2, 3};
v1.resize(5, 0);
for(int i = 0; i < v1.size(); i++){
    cout << v1[i] << " ";
}
//1 2 3 0 0
vector<int> v2 = {1, 2, 3, 4, 5};
v2.resize(3, 0);
for(int i = 0; i < v2.size(); i++){
    cout << v2[i] << " ";
}
//1 2 3

v.max_size()
返回动态数组 v 可以存储元素的最大数量 (假设内存无限)O(1)
vector<int> v;
cout << v.max_size();

v.capacity()
返回动态数组 v 当前的容量 (内存不够会两倍两倍的加)O(1)
vector<int> v;
for(int i = 0; i < 6; i++){
    cout << v.capacity() << " ";
    v.emplace_back(i);
}
//0 1 2 4 4 8

v.reserve()
更改动态数组 v 的当前的容量 (capacity)O(1)
vector<int> v;
cout << v.capacity() << " ";
```

```
v.reserve(10);
cout << v.capacity() << " ";
//0 10
```

```
vector<int> v;
```
开一个没有任何元素的一维动态数组 v

```
vector<int> v(n);
```
开一个长度为 n 的一维动态数组 v，初始值默认为 0

```
vector<int> v(n, x);
```
开一个长度为 n 的一维动态数组 v，初始值默认为 x

```
vector<vector<int>> v;
```
开一个没有任何元素的二维动态数组 v

```
vector<vector<int>> v(n);
```
开一个一维长度为 n 的二维动态数组

```
vector<vector<int>> v(n, vector<int> (m));
```
开一个 n * m 的二维动态数组，初始值默认为 0

```
vector<vector<int>> v(n, vector<int> (m, x));
```
开一个 n * m 的二维动态数组，初始值默认为 x

```
vector<int> v[N];
```
开一个一维长度固定为 N 的二维动态数组

```
vector<int> v;
v = vector<int> ();
```
开一个没有任何元素的一维动态数组 v

```
vector<int> v;
v = vector<int> (n);
```
开一个没有任何元素的一维动态数组 v

```
vector<int> v;
v = vector<int> (n, x);
```

开一个长度为 n 的一维动态数组 v，初始值默认为 0

```
vector<vector<int>> v;
v = vector<vector<int>> ();
```

开一个没有任何元素的二维动态数组 v

```
vector<vector<int>> v;
v = vector<vector<int>> (n);
```

开一个一维长度为 n 的二维动态数组

```
vector<vector<int>> v;
v = vector<vector<int>> (n, vector<int> (m));
```

开一个 n * m 的二维动态数组，初始值默认为 0

```
vector<vector<int>> v;
v = vector<vector<int>> (n, vector<int> (m, x));
```

开一个 n * m 的二维动态数组，初始值默认为 x

```
vector<int> a = {1, 2, 3, 4};
vector<int> b(a);
vector<int> c = a;
```

a 是拷贝初始化

b 是拷贝初始化

c 是拷贝初始化

a = b = c

2 stack 栈

```
stack<element_type> stk; 栈

stk.size()
返回栈 stk 的元素的个数 (返回类型为 usinged) O(1)
stack<int> stk;
cout << stk.size() << " ";
stk.push(0);
cout << stk.size() << " ";
//0 1

stk.empty()
判断栈 stk 是否为空
空则返回 true, 非空则返回 false (返回类型是 bool) O(1)
stack<int> stk;
cout << stk.empty() << " ";
stk.push(0);
cout << stk.empty() << " ";
//1 0

stk.top()
返回栈 stk 顶的元素 O(1)
stack<int> stk;
stk.push(1);
cout << stk.top() << " ";
stk.push(4);
cout << stk.top() << " ";
//1 4

stk.push(element)
添加一个元素 element 在栈 stk 顶 O(1)
stack<int> stk;
stk.push(1);
//stk = {1};

stk.emplace(element)
```

添加一个元素 element 在栈 stk 顶 O(1)

```
stack<int> stk;
stk.emplace(1);
//stk = {1};
```

stk.pop()

删除栈 stk 顶的元素 O(1)

```
stack<int> stk;
stk.push(1);
stk.push(2);
//stk = {1, 2}
stk.pop();
//stk = {1}
```

3 queue 队列

```
queue<element_type> q; 队列

q.size()
返回队列 q 的元素的个数 (返回类型为 usinged) O(1)
queue<int> q;
cout << q.size() << endl; // 0
q.push(1), q.push(2);
cout << q.size() << endl; // 2

q.empty()
判断队列 q 是否为空
空则返回 true, 非空则返回 false (返回类型是 bool) O(1)
queue<int> q;
cout << q.empty() << endl; // 1
q.push(1), q.push(2);
cout << q.empty() << endl; // 0

q.front()
返回队列 q 的第一个元素 O(1)
queue<int> q;
q.push(1), q.push(2);
cout << q.front(); // 1

q.back()
返回队列 q 的最后一个元素 O(1)
queue<int> q;
q.push(1), q.push(2);
cout << q.back(); // 2

q.push(element)
添加一个元素 element 在队列 q 的最后面 O(1)
queue<int> q;
q.push(1), q.push(2);
//q = {1, 2}
```

q.emplace(element)
添加一个元素 element 在队列 q 的最后面 O(1)

```
queue<int> q;
q.emplace(1), q.emplace(2);
//q = {1, 2}
```

q.pop()
删除队列 q 的第一个元素 O(1)

```
queue<int> q;
q.push(1), q.push(2);
//q = {1, 2}
q.pop();
//q = {2}
```

4 deque 双端队列

```
deque<element_type> dq; 双端队列
```

```
dq.size()
返回双端队列 dq 的元素的个数 (返回类型为 usinged) O(1)
deque<int> dq1 = {1, 2, 3, 4};
cout << dq1.size(); // 4
deque<int> dq2 = {};
cout << dq2.size(); // 0
```

```
dq.empty()
判断双端队列 dq 是否为空
空则返回 true, 非空则返回 false(返回类型是 bool)O(1)
deque<int> dq1 = {1, 2, 3, 4};
cout << dq1.empty(); // 0
deque<int> dq2 = {};
cout << dq2.empty(); // 1
```

```
dq.front()
返回双端队列 dq 的第一个元素 O(1)
deque<int> dq1 = {1, 2, 3, 4};
cout << dq1.front(); // 1
deque<int> dq2 = {4, 3, 2, 1};
cout << dq2.front(); // 4
```

```
dq.back()
返回双端队列 dq 的最后一个元素 O(1)
deque<int> dq1 = {1, 2, 3, 4};
cout << dq1.front(); // 4
deque<int> dq2 = {4, 3, 2, 1};
cout << dq2.front(); // 1
```

```
dq.begin()
返回指向双端队列 dq 第一个元素的迭代器 (正向) O(1)
deque<int> dq1 = {1, 2, 3, 4};
```

```
cout << *dq1.begin(); // 1
deque<int> dq2 = {4, 3, 2, 1};
cout << *dq2.begin(); // 4
```

dq.end()
返回指向双端队列 dq 的最后一个元素的后一个位置的迭代器（正向）O(1)

dq.rbegin()
返回指向双端队列 dq 的最后一个元素的迭代器（逆向）O(1)

```
deque<int> dq1 = {1, 2, 3, 4};
cout << *dq1.rbegin(); // 4
deque<int> dq2 = {4, 3, 2, 1};
cout << *dq2.rbegin(); // 1
```

dq.rend()
返回指向双端队列 dq 的第一个元素的前一个位置的迭代器（逆向）O(1)

dq.push_back(element)^^I
添加一个元素在双端队列 dq 的最后面 O(1)

```
deque<int> dq = {1, 2, 3};
dq.push_back(5);
// dq = {1, 2, 3, 5}
dq.push_back(4);
// dq = {1, 2, 3, 5, 4}
```

dq.emplace_back(element)
添加一个元素在双端队列 dq 的最后面 O(1)

```
deque<int> dq = {1, 2, 3};
dq.emplace_back(5);
// dq = {1, 2, 3, 5}
dq.emplace_back(4);
// dq = {1, 2, 3, 5, 4}
```

dq.push_front(element)
添加一个元素在双端队列 dq 的最前面 O(1)

```
deque<int> dq = {1, 2, 3};
dq.push_front(5);
```

```
// dq = {5, 1, 2, 3}
dq.push_front(4);
// dq = {4, 5, 1, 2, 3}
```

dq.emplace_front(element)
添加一个元素在双端队列 dq 的最前面 O(1)

```
deque<int> dq = {1, 2, 3};
dq.emplace_front(5);
// dq = {5, 1, 2, 3}
dq.emplace_front(4);
// dq = {4, 5, 1, 2, 3}
```

dq.insert(it, x)
在迭代器 it 处插入一个元素 x O(n)

```
deque<int> dq = {1, 2, 3};
dq.insert(dq.begin() + 1, 4);
//dq = {1, 4, 2, 3}
```

dq.pop_front()
删除双端队列 dq 的第一个元素 O(1)

```
deque<int> dq = {1, 2, 3};
dq.pop_front();
//dq = {2, 3}
```

dq.pop_back()
删除双端队列 dq 的最后一个元素 O(1)

```
deque<int> dq = {1, 2, 3};
dq.pop_back();
//dq = {1, 2}
```

dq.erase(it)
删除双端队列 dq 迭代器 it 指向的元素 O(n)

```
deque<int> dq = {1, 2, 3};
dq.erase(dq.begin() + 1);
//dq = {1, 3}
```

dq.erase(first,last)

删除双端队列 dq 迭代器 [first, last) 指向的元素 O(n)

```
deque<int> dq = {1, 2, 3};
dq.erase(dq.begin() + 1, dq.begin() + 2);
//dq = {1, 3}
```

dq.clear()

清除双端队列的所有元素

```
deque<int> dq = {1, 2, 3};
dq.clear();
//dq = {}
```

dq.max_size()

返回双端队列 dq 可以存储元素的最大数量 (假设内存无限)O(1)

```
deque<int> dq;
cout << dq.max_size();
```

5 priority_queue 优先队列

```
priority_queue<element_type> q; 优先队列

q.size()
返回优先队列 q 的元素的个数 (返回类型为 usinged) O(1)
priority_queue<int> q;
q.push(1), q.push(2), q.push(3);
cout << q.size(); // 3

q.empty()
判断优先队列 q 是否为空
空则返回 true, 非空则返回 false(返回类型是 bool) O(1)
priority_queue<int> q;
cout << q.empty(); // 1
q.push(1), q.push(2), q.push(3);
cout << q.empty(); // 0

q.top()
返回优先队列 q 中优先级最高的元素 O(1)
priority_queue<int> q;
q.push(1), q.push(2), q.push(3);
cout << q.top(); // 3

q.push(element)
添加一个元素 element 在优先队列 q 中 O(logn)
priority_queue<int> q;
q.push(4), q.push(2), q.push(3);
cout << q.top(); // 4

q.emplace(element)
添加一个元素 element 在优先队列 q 中 O(logn)
priority_queue<int> q;
q.emplace(4), q.emplace(2), q.emplace(3);
cout << q.top(); // 4

q.pop()
```

删除优先队列 q 中优先级最高的元素 O(1)

```
priority_queue<int> q;
q.emplace(4), q.emplace(2), q.emplace(3);
cout << q.top(); // 4
q.pop();
cout << q.top(); // 3
```

6 map 有序 键值唯一 哈希表

```
map<element_type1,element_type2> mp; 哈希表
mp.size()
返回哈希表 mp 的元素的个数 (返回类型为 usinged) O(1)
map<int, int> mp1 = {{0, 0}, {1, 0}};
cout << mp1.size(); // 2;
map<int, int> mp2 = {{0, 0}, {1, 0}, {2, 0}};
cout << mp2.size(); // 3;

mp.empty()
判断哈希表 mp 是否为空
空则返回 true, 非空则返回 false (返回类型是 bool) O(1)
map<int, int> mp1 = {{0, 0}, {1, 0}};
cout << mp1.empty(); // 0;
map<int, int> mp2 = {};
cout << mp2.empty(); // 1;

mp.begin()
返回指向哈希表 mp 第一个元素的迭代器 O(1)
map<int, int> mp1 = {{2, 0}, {3, 0}};
cout << mp1.begin()->first; // 2;
map<int, int> mp2 = {{4,0}, {5, 0}};
cout << mp2.begin()->first; // 4;

mp.end()
返回指向哈希表 mp 最后一个元素的后一个位置的迭代器 O(1)

mp.rbegin()
返回指向哈希表 mp 最后一个元素的迭代器 O(1)
map<int, int> mp1 = {{2, 0}, {3, 0}};
cout << mp1.rbegin()->first; // 3;
map<int, int> mp2 = {{4,0}, {5, 0}};
cout << mp2.rbegin()->first; // 5;

mp.rend()
返回指向哈希表 mp 第一个元素的前一个位置的迭代器 O(1)
```

```
mp.insert({element_type1, element_type2})
在哈希表 mp 中插入键值对 {element_type1, element_type2} O(logn)
map<int, int> mp;
//mp = {}
mp.insert({10, 20});
//mp = {{10, 20}}
mp.insert({20, 20});
//mp = {{10, 20}, {20, 20}}

mp.emplace(element_type1, element_type2)
在哈希表 mp 中插入键值对 {element_type1, element_type2} O(logn)
map<int, int> mp;
//mp = {}
mp.emplace(10, 20);
//mp = {{10, 20}}
mp.emplace(20, 20);
//mp = {{10, 20}, {20, 20}}

mp.erase(it)
删除哈希表 mp 中迭代器 it 指向的键值对 O(logn)
map<int, int> mp;
//mp = {}
mp.insert({10, 20});
//mp = {{10, 20}}
mp.insert({20, 20});
//mp = {{10, 20}, {20, 20}}
mp.erase(mp.begin());
//mp = {{20, 20}}

mp.erase(key)
删除哈希表 mp 中键值为 key 的键值对 O(logn)
map<int, int> mp;
//mp = {}
mp.insert({10, 20});
//mp = {{10, 20}}
mp.insert({20, 20});
```

```
//mp = {{10, 20}, {20, 20}}
mp.erase(10);
//mp = {{20, 20}}

mp.erase(first,last)
删除哈希表 mp 中迭代器 [first, last) 之间的键值对 O(logn)
map<int, int> mp;
//mp = {}
mp.insert({10, 20});
//mp = {{10, 20}}
mp.insert({20, 20});
//mp = {{10, 20}, {20, 20}}
mp.erase(mp.begin(), mp.end());
//mp = {}

mp.clear()
清除哈希表 mp 中所有的键值对 O(n)
map<int, int> mp;
//mp = {}
mp.insert({10, 20});
//mp = {{10, 20}}
mp.insert({20, 20});
//mp = {{10, 20}, {20, 20}}
mp.clear();
//mp = {}

mp.find(key)
查找哈希表 mp 中有没有键值为 key 的键值对
有就返回指向该键值对的迭代器 it, 没有就返回 mp.end() O(logn)
map<int, int> mp ={{1, 2}, {2, 3}};
cout << mp.find(1)->second; // 2;
cout << (mp.find(3) == mp.end()); // 1

mp.count(key)
返回哈希表 mp 中键值 key 出现的次数 O(logn)
map<int, int> mp = {{1, 2}, {2, 3}, {3, 4}};
cout << mp.count(1); // 1;
```

```
cout << mp.count(5); // 0

mp.lower_bound()
返回哈希表 mp 中键值 >= key 的迭代器，没有就返回 mp.end() O(logn)
map<int, int> mp = {{1, 2}, {2, 3}, {3, 4}};
cout << mp.lower_bound(1)->first; // 1

mp.upper_bound()
返回哈希表 mp 中键值 > key 的迭代器，没有就返回 mp.end() O(logn)
map<int, int> mp = {{1, 2}, {2, 3}, {3, 4}};
cout << mp.upper_bound(1)->first; // 2

mp.max_size()
返回哈希表 mp 的最大存储键值对的个数（假设空间无限）O(1)
map<int, int> mp;
cout << mp.max_size();
```

7 unordered_map 无序 键值唯一 哈希表

```
unordered_map<element_type1,element_type2> mp; 哈希表
mp.size()
返回哈希表 mp 的元素的个数 (返回类型为 usinged) O(1)
unordered_map<int, int> mp1 = {{0, 0}, {1, 0}};
cout << mp1.size(); // 2;
unordered_map<int, int> mp2 = {{0, 0}, {1, 0}, {2, 0}};
cout << mp2.size(); // 3;

mp.empty()
判断哈希表 mp 是否为空
空则返回 true, 非空则返回 false (返回类型是 bool) O(1)
unordered_map<int, int> mp1 = {{0, 0}, {1, 0}};
cout << mp1.empty(); // 0;
unordered_map<int, int> mp2 = {};
cout << mp2.empty(); // 1;

mp.begin()
返回指向哈希表 mp 第一个元素的迭代器 O(1)
mp.end()
返回指向哈希表 mp 最后一个元素的后一个位置的迭代器 O(1)
mp.rbegin()
返回指向哈希表 mp 最后一个元素的迭代器 O(1)
mp.rend()
返回指向哈希表 mp 第一个元素的前一个位置的迭代器 O(1)

mp.insert({element_type1, element_type2})
在哈希表 mp 中插入键值对 {element_type1, element_type2} O(1) / O(n)
unordered_map<int, int> mp;
//mp = {}
mp.insert({10, 20});
//mp = {{10, 20}}
mp.insert({20, 20});
//mp = {{10, 20}, {20, 20}}

mp.emplace(element_type1, element_type2)
```

```
在哈希表 mp 中插入键值对 {element_type1, element_type2} O(1) / O(n)
unordered_map<int, int> mp;
//mp = {}
mp.emplace(10, 20);
//mp = {{10, 20}}
mp.emplace(20, 20);
//mp = {{10, 20}, {20, 20}}

mp.erase(it)
删除哈希表 mp 中迭代器 it 指向的键值对 O(1) / O(n)
unordered_map<int, int> mp;
//mp = {}
mp.insert({10, 20});
//mp = {{10, 20}}
mp.insert({20, 20});
//mp = {{10, 20}, {20, 20}}
mp.erase(mp.find(10));
//mp = {{20, 20}}

mp.erase(key)
删除哈希表 mp 中键值为 key 的键值对 O(1) / O(n)
unordered_map<int, int> mp;
//mp = {}
mp.insert({10, 20});
//mp = {{10, 20}}
mp.insert({20, 20});
//mp = {{10, 20}, {20, 20}}
mp.erase(10);
//mp = {{20, 20}}

mp.clear()
清除哈希表 mp 中所有的键值对 O(n)
unordered_map<int, int> mp;
//mp = {}
mp.insert({10, 20});
//mp = {{10, 20}}
mp.insert({20, 20});
```

```
//mp = {{10, 20}, {20, 20}}
mp.clear();
//mp = {}

mp.find(key)
查找哈希表 mp 中有没有键值为 key 的键值对
有就返回指向该键值对的迭代器 it, 没有就返回 mp.end() O(1) / O(n)
unordered_map<int, int> mp = {{1, 2}, {2, 3}};
cout << mp.find(1)->second; // 2
cout << (mp.find(3) == mp.end()); // 1

mp.count(key)
返回哈希表 mp 中键值 key 出现的次数 O(1) / O(n)
unordered_map<int, int> mp ={{1, 2}, {2, 3}, {3, 4}};
cout << mp.count(1); // 1
cout << mp.count(5); // 0

mp.max_size()
返回哈希表 mp 的最大存储键值对的个数 (假设空间无限) O(1)
unordered_map<int, int> mp;
cout << mp.max_size();
```

8 set 有序 键值唯一 集合

```
set<element_type> s; 集合

s.size()
返回集合 s 的元素的个数 (返回类型为 usinged) O(1)
set<int> s1 = {1, 2, 3, 4};
cout << s1.size(); // 4
set<int> s2 = {5, 6, 7};
cout << s2.size(); // 3

s.empty()
判断集合 s 是否为空
空则返回 true, 非空则返回 false(返回类型是 bool) O(1)
set<int> s1 = {1, 2, 3, 4};
cout << s1.empty(); // 0
set<int> s2 = {};
cout << s2.empty(); // 1

s.begin()
返回指向集合 s 第一个元素的迭代器 O(1)
set<int> s1 = {1, 2, 3, 4};
cout << *s1.begin(); // 1
set<int> s2 = {5, 6, 7};
cout << *s2.begin(); // 5

s.end()
返回指向集合 s 最后一个元素的后一个位置的迭代器 O(1)

s.rbegin()
返回指向集合 s 最后一个元素的迭代器 O(1)
set<int> s1 = {1, 2, 3, 4};
cout << *s1.rbegin(); // 4
set<int> s2 = {5, 6, 7};
cout << *s2.rbegin(); // 7

s.rend()
```

返回指向集合 s 第一个元素的前一个位置的迭代器 O(1)

s.insert(element)
在集合 s 中插入一个元素 element O(logn)

```
set<int> s;
//s = {}
s.insert(2);
//s = {2}
s.insert(2);
//s = {2}
s.insert(3);
//s = {2, 3}
```

s.emplace(element)
在集合 s 中插入一个元素 element O(logn)

```
set<int> s;
//s = {}
s.emplace(2);
//s = {2}
s.emplace(2);
//s = {2}
s.emplace(3);
//s = {2, 3}
```

s.erase(it)
删除集合 s 中迭代器 it 所指向的元素 O(logn)

```
set<int> s = {1, 2, 3, 4};
s.earse(++s.begin());
//s = {1, 3, 4};
```

s.erase(first,last)
删除集合 s 中迭代器 [first, last) 之间的元素 O(logn)

```
set<int> s = {1, 2, 3, 4};
s.erase(s.begin(), s.end());
//s = {};
```

s.erase(key)

删除集合 s 中值为 key 的元素 O(logn)

```
set<int> s = {1, 2, 3, 4};
s.erase(2);
//s = {1, 3, 4};
```

s.extract(key)

```
set<int> s = {1, 2, 3, 4};
auto it = s.extract(2);
// s = {1, 3, 4}
```

s.clear()

清除集合 s 的所有元素 O(n)

```
set<int> s = {1, 2, 3, 4};
s.clear();
//s = {};
```

s.find(element)

查找集合 s 中有没有值为 element

有就返回指向该键值对的迭代器 it，没有就返回 s.end() O(logn)

```
set<int> s = {1, 2, 3, 4};
cout << *s.find(3); // 3
cout << (s.find(5) == s.end()); // 1
```

s.count(element)

查找集合 s 中值为 element 的元素个数 O(logn)

```
set<int> s = {1, 2, 3, 4};
cout << s.count(3); // 1
cout << s.count(5); // 0
```

s.lower_bound(k)

返回集合 s 中值 >= key 的迭代器，没有就返回 s.end() O(logn)

```
set<int> s = {1, 2, 3, 4};
cout << *s.lower_bound(3); // 3
```

s.upper_bound(k)

返回集合 s 中键值 > key 的迭代器，没有就返回 s.end() O(logn)

```
set<int> s = {1, 2, 3, 4};
```

```
cout << *s.upper_bound(3); // 4
```

9 unordered_set 无序 键值唯一 集合

```
unordered_set<element_type> s; 集合
s.size()
返回集合 s 的元素的个数 (返回类型为 usinged) O(1)
unordered_set<int> s1 = {1, 2, 3, 4};
cout << s1.size(); // 4
unordered_set<int> s2 = {5, 6, 7};
cout << s2.size(); // 3

s.empty()
判断集合 s 是否为空
空则返回 true, 非空则返回 false(返回类型是 bool) O(1)
unordered_set<int> s1 = {1, 2, 3, 4};
cout << s1.empty(); // 0
unordered_set<int> s2 = {};
cout << s2.empty(); // 1

s.begin()
返回指向集合 s 第一个元素的迭代器 O(1)

s.end()
返回指向集合 s 最后一个元素的后一个位置的迭代器 O(1)

s.rbegin()
返回指向集合 s 最后一个元素的迭代器 O(1)

s.rend()
返回指向集合 s 第一个元素的前一个位置的迭代器 O(1)

s.insert(element)
在集合 s 中插入一个元素 element O(1) / O(n)
unordered_set<int> s;
//s = {}
s.insert(2);
//s = {2}
s.insert(2);
```

```
//s = {2}
s.insert(3);
//s = {2, 3}

s.emplace(element)
在集合 s 中插入一个元素 element O(1) / O(n)
unordered_set<int> s;
//s = {}
s.emplace(2);
//s = {2}
s.emplace(2);
//s = {2}
s.emplace(3);
//s = {2, 3}

s.erase(it)
删除集合 s 中迭代器 it 所指向的元素 O(n)

s.erase(first,last)
删除集合 s 中迭代器 [first, last) 之间的元素 O(n)

s.erase(key)
删除集合 s 中值为 key 之间的元素 O(1)/O(n)
unordered_set<int> s = {1, 2, 3, 4};
s.erase(2);
//s = {1, 3, 4};

s.clear()
清除集合 s 的所有元素 O(n)
unordered_set<int> s = {1, 2, 3, 4};
s.clear();
//s = {};

s.find(element)
查找集合 s 中有没有值为 element
有就返回指向该键值对的迭代器 it, 没有就返回 s.end() O(1) / O(n)
unordered_set<int> s = {1, 2, 3, 4};
```

```
cout << *s.find(3); // 3
cout << (s.find(5) == s.end()); // 1

s.count(element)
查找集合 s 中值为 element 的元素个数 O(1) / O(n)
unordered_set<int> s = {1, 2, 3, 4};
cout << s.count(3); // 1
cout << s.count(5); // 0
```

10 multimap 有序 键值不唯一 哈希表

```
multimap<element_type1,element_type2> mp;
```

```
mp.size()
返回哈希表 mp 的元素的个数 (返回类型为 usinged) O(1)
multimap<int, int> mp1 = {{0, 0}, {1, 0}};
cout << mp1.size(); // 2;
multimap<int, int> mp2 = {{0, 0}, {1, 0}, {2, 0}};
cout << mp2.size(); // 3;
```

```
mp.empty()
判断哈希表 mp 是否为空
空则返回 true, 非空则返回 false(返回类型是 bool) O(1)
multimap<int, int> mp1 = {{0, 0}, {1, 0}};
cout << mp1.empty(); // 0;
multimap<int, int> mp2 = {};
cout << mp2.empty(); // 1;
```

```
mp.begin()
返回指向哈希表 mp 第一个元素的迭代器 O(1)
multimap<int, int> mp1 = {{2, 0}, {3, 0}};
cout << mp1.begin()->first; // 2;
multimap<int, int> mp2 = {{4, 0}, {5, 0}};
cout << mp2.begin()->first; // 4;
```

```
mp.end()
返回指向哈希表 mp 最后一个元素的后一个位置的迭代器 O(1)
```

```
mp.rbegin()
返回指向哈希表 mp 最后一个元素的迭代器 O(1)
multimap<int, int> mp1 = {{2, 0}, {3, 0}};
cout << mp1.rbegin()->first; // 3;
multimap<int, int> mp2 = {{4, 0}, {5, 0}};
cout << mp2.rbegin()->first; // 5;
```

```
mp.rend()
```

返回指向哈希表 mp 第一个元素的前一个位置的迭代器 O(1)

mp.insert({element_type1, element_type2})
在哈希表 mp 中插入键值对 {element_type1, element_type2} O(logn)

```
multimap<int, int> mp;
//mp = {}
mp.insert({10, 20});
//mp = {{10, 20}}
mp.insert({20, 20});
//mp = {{10, 20}, {20, 20}}
mp.insert({20, 20});
//mp = {{10, 20}, {20, 20}, {20, 20}}
cout << mp.size() << endl; // 3
```

mp.emplace(element_type1, element_type2)
在哈希表 mp 中插入键值对 {element_type1, element_type2} O(logn)

```
multimap<int, int> mp;
//mp = {}
mp.emplace(10, 20);
//mp = {{10, 20}}
mp.emplace(20, 20);
//mp = {{10, 20}, {20, 20}}
mp.emplace(20, 20);
//mp = {{10, 20}, {20, 20}, {20, 20}}
cout << mp.size() << endl; // 3
```

mp.erase(it)
删除哈希表 mp 中迭代器 it 指向的键值对 O(logn)

```
multimap<int, int> mp;
//mp = {}
mp.insert({10, 20});
//mp = {{10, 20}}
mp.insert({20, 20});
//mp = {{10, 20}, {20, 20}}
mp.erase(mp.begin());
//mp = {{20, 20}}
```

```
mp.erase(key)
删除哈希表 mp 中键值为 key 的键值对 O(logn)
multimap<int, int> mp;
//mp = {}
mp.insert({10, 20});
//mp = {{10, 20}}
mp.insert({20, 20});
//mp = {{10, 20}, {20, 20}}
mp.insert({20, 20});
//mp = {{10, 20}, {20, 20}, {20, 20}}
mp.erase(20);
//mp = {{10, 20}}

mp.erase(first,last)
删除哈希表 mp 中迭代器 [first, last) 之间的键值对 O(logn)
multimap<int, int> mp;
//mp = {}
mp.insert({10, 20});
//mp = {{10, 20}}
mp.insert({20, 20});
//mp = {{10, 20}, {20, 20}}
mp.erase(mp.begin(), mp.end());
//mp = {}

mp.clear()
清除哈希表 mp 中所有的键值对 O(n)
multimap<int, int> mp;
//mp = {}
mp.insert({10, 20});
//mp = {{10, 20}}
mp.insert({20, 20});
//mp = {{10, 20}, {20, 20}}
mp.clear();
//mp = {}

mp.find(key)
查找哈希表 mp 中有没有键值为 key 的键值对
```

有就返回指向该键值对的迭代器 it，没有就返回 mp.end() O(logn)

```
multimap<int, int> mp ={{1, 2}, {2, 3}};
cout << mp.find(1)->second; // 2;
cout << (mp.find(3) == mp.end()); // 1
```

mp.count(key)

返回哈希表 mp 中键值 key 出现的次数 O(logn)

```
multimap<int, int> mp ={{1, 2}, {2, 3}, {3, 4}, {3, 5}};
cout << mp.count(1); // 1;
cout << mp.count(3); // 2
```

mp.lower_bound()

返回哈希表 mp 中键值 >= key 的迭代器，没有就返回 mp.end() O(logn)

```
multimap<int, int> mp ={{1, 2}, {2, 3}, {3, 4}};
cout << mp.lower_bound(1)->first; // 1
```

mp.upper_bound()

返回哈希表 mp 中键值 > key 的迭代器，没有就返回 mp.end() O(logn)

```
multimap<int, int> mp ={{1, 2}, {2, 3}, {3, 4}};
cout << mp.upper_bound(1)->first; // 2
```

mp.max_size()

返回哈希表 mp 的最大存储键值对的个数（假设空间无限）O(1)

```
multimap<int, int> mp;
cout << mp.max_size();
```

11 multiset 有序 键值不唯一 集合

multiset<element_type> s; 集合 (有顺序)

s.size()
返回集合 s 的元素的个数 (返回类型为 usinged) O(1)

```
set<int> s1 = {1, 2, 3, 4};
cout << s1.size(); // 4
set<int> s2 = {5, 6, 7};
cout << s2.size(); // 3
```

s.empty()
判断集合 s 是否为空, 空则返回 true, 非空则返回 false(返回类型是 bool) O(1)

```
set<int> s1 = {1, 2, 3, 4};
cout << s1.empty(); // 0
set<int> s2 = {};
cout << s2.empty(); // 1
```

s.begin()
返回指向集合 s 第一个元素的迭代器 O(1)

```
set<int> s1 = {1, 2, 3, 4};
cout << *s1.begin(); // 1
set<int> s2 = {5, 6, 7};
cout << *s2.begin(); // 5
```

s.end()
返回指向集合 s 最后一个元素的后一个位置的迭代器 O(1)

s.rbegin()
返回指向集合 s 最后一个元素的迭代器 O(1)

```
set<int> s1 = {1, 2, 3, 4};
cout << *s1.rbegin(); // 4
set<int> s2 = {5, 6, 7};
cout << *s2.rbegin(); // 7
```

s.rend()
返回指向集合 s 第一个元素的前一个位置的迭代器 O(1)

s.insert(element)
在集合 s 中插入一个元素 element O(logn)

```
set<int> s;
//s = {}
s.insert(2);
//s = {2}
s.insert(2);
//s = {2, 2}
s.insert(3);
//s = {2, 2, 3}
```

s.emplace(element)
在集合 s 中插入一个元素 element O(logn)

```
set<int> s;
//s = {}
s.emplace(2);
//s = {2}
s.emplace(2);
//s = {2, 2}
s.emplace(3);
//s = {2, 2, 3}
```

s.erase(it)
删除集合 s 中迭代器 it 所指向的元素 O(logn)

```
set<int> s = {1, 2, 3, 4};
s.erase(++s.begin());
//s = {1, 3, 4};
```

s.erase(first,last)
删除集合 s 中迭代器 [first, last) 之间的元素 O(logn)

```
set<int> s = {1, 2, 3, 4};
s.erase(s.begin(), s.end());
//s = {};
```

s.erase(key)
删除集合 s 中值为 key 之间的元素 O(logn)

```cpp
set<int> s = {1, 2, 3, 4};
s.erase(2);
//s = {1, 3, 4};

s.clear()
清楚集合 s 的所有元素 O(n)
set<int> s = {1, 2, 3, 4};
s.clear();
//s = {};

s.find(element)
查找集合 s 中有没有值为 element
有就返回指向该键值对的迭代器 it, 没有就返回 s.end() O(logn)
set<int> s = {1, 2, 3, 4};
cout << *s.find(3); // 3
cout << (s.find(5) == s.end()); // 1

s.count(element)
查找集合 s 中值为 element 的元素个数 O(logn)
set<int> s = {1, 2, 3, 3, 4};
cout << s.count(3); // 2
cout << s.count(5); // 0

s.lower_bound(k)
返回集合 s 中值 >= key 的迭代器, 没有就返回 s.end() O(logn)
set<int> s = {1, 2, 3, 4};
cout << *s.lower_bound(3); // 3

s.upper_bound(k)
返回集合 s 中键值 > key 的迭代器, 没有就返回 s.end() O(logn)
set<int> s = {1, 2, 3, 4};
cout << *s.upper_bound(3); // 4
```

12 pair 键值对

pair<element_type1,element_type2> p;
(相当于有两个元素的结构体，能比较大小，按照 first 和 second 的顺序比较)

make_pair{element1, element2};
创造一个 pair

13 string 字符串

```
string s; 字符串
s.size()/s.length()
返回字符串 s 中字符的个数 O(1)
string s1 = "abc";
cout << s1.size(); // 3
cout << s1.length(); // 3
string s2 = "abcd";
cout << s2.size(); // 4
cout << s2.length(); // 4

s.empty()
判断字符串 s 是否为空
空则返回 true，非空则返回 false(返回类型是 bool) O(1)
string s1 = "abc";
cout << s1.empty(); //0
string s2 = "";
cout << s2.empty(); //1

s.front()
返回字符串 s 的第一个字符 O(1)
string s1 = "abcd";
string s2 = "bcda";
cout << s1.front(); //a
cout << s2.front(); //b

s.back()
返回字符串 s 的最后一个字符 O(1)
string s1 = "abcd";
string s2 = "bcda";
cout << s1.back(); //d
cout << s2.back(); //a

s.begin()
返回指向字符串 s 第一个字符的迭代器（正向）O(1)
string s1 = "abcd";
```

```
string s2 = "bcda";
cout << *s1.begin(); //a
cout << *s2.begin(); //b
```

s.end()
返回指向字符串 s 的最后一个字符的后一个位置的迭代器（正向）O(1)

```
s.rbegin()
```
返回指向字符串 s 的最后一个字符的迭代器（逆向）O(1)
```
string s1 = "abcd";
string s2 = "bcda";
cout << *s1.rbegin(); //d
cout << 8s2.rbegin(); //a
```

s.rend()
返回指向字符串 s 的第一个字符的前一个位置的迭代器（逆向）O(1)

s.push_back(element)
添加一个字符 element 在字符串 s 的最后面 O(1)
```
string s = "123";
s.push_back('1');
cout << s << endl;
//s = "1231"
```

s.insert(pos, str)
插入一个字符串 str 在字符串 s 的 pos 位置 O(n)
```
string s = "123";
s.insert(1, "2");
cout << s << endl;
//s = "1223"
```

s.append(str)
在字符串 s 后面添加一个字符串 str 在最后面 O(n)
```
string s = "123";
s.append("456");
cout << s << endl;
//s = "123456"
```

```
s.erase(it)
删除字符串 s 中迭代器 it 所指的字符 O(n)
string s = "123456";
s.erase(s.begin() + 3);
cout << s << endl;
//s = "12356"

s.erase(first, last)
删除字符串 s 中迭代器区间 [first,last) 上所有字符 O(n)
string s = "123456";
s.erase(s.begin() + 2, s.begin() + 4);
cout << s << endl;
//s = "1256"

s.erase(pos, len)
删除字符串 s 中从索引位置 pos 开始的 len 个字符 O(n)
string s = "123456";
s.erase(2, 2);
cout << s << endl;
//s = "1256"

s.clear()
清除字符串 s 中所有字符 O(n)
string s = "123456";
s.clear();
cout << s << endl;
//s = ""

s.max_size()
返回字符串 s 可以存储字符的最大数量 (假设内存无限)O(1)
string s;
cout << s.max_size();

s.capacity()
返回字符串 s 当前的容量 (内存不够会两倍两倍的加)O(1)
string s = "1231";
```

```
cout << capacity();
```

s.replace(pos, n, str)
把当前字符串 s 从索引 pos 开始的 n 个字符替换为 str O(n)

```
string s = "0123456789";
s.replace(1, 3, "abc");
cout << s << endl;
//s = "0abc456789"
```

s.replace(pos, x, y, c)
把当前字符串 s 从索引 pos 开始的 x 个字符替换为 y 个字符 c O(n)

```
string s = "0123456789";
s.replace(1, 3, 3, 'c');
//s = "0ccc456789"
```

s.replace(first, last, str)
把当前字符串 s 迭代器 [first, last) 之间替换为 str O(n)

```
string s = "0123456789";
s.replace(s.begin() + 1, s.begin() + 3, "abc");
//s = "0abc456789"
```

tolower(s[i])
转换为小写 O(1)

```
char ch = 'A';
ch = tolower(ch);
//ch = 'a'
```

toupper(s[i])
转换为大写 O(1)

```
char ch = 'a';
ch = tolower(ch);
//ch = 'A'
```

transform(s.begin(),s.end(),s.begin(),::tolower);
转换小写 O(n)

```
transform(s.begin(),s.end(),s.begin(),::toupper);
```

转换大写 O(n)

s.substr(pos, n)
截取从 pos 索引开始的 n 个字符 O(n)

```
string s1 = "0123456789";
string s2 = s1.substr(1, 3);
string s3 = s1.substr(1);
//s2 = "123"
//s3 = "123456789"
```

s.find(str, pos)
在当前字符串 s 的 pos 索引位置 (默认为 0) 开始, 查找子串 str
返回找到的位置索引, -1表示查找不到子串 O(n)

```
string s = "0123456789";
cout << s.find("987") << endl; // 1
cout << (long long)s.find("789") << endl; // -1(无符号整数)
cout << s.find("987", 0) << endl; // 1
cout << (long long)s.find("789", 0) << endl; // -1(无符号整数)
```

s.find(c, pos)
在当前字符串 s 的 pos 索引位置 (默认为 0) 开始, 查找字符 c
返回找到的位置索引, -1表示查找不到字符 O(n)

```
string s = "0123456789";
cout << s.find('1'); // 1
cout << (long long)s.find('a'); // -1(无符号整数)
cout << s.find('1', 0); // 1
cout << (long long)s.find('a', 0); // -1(无符号整数)
```

s.rfind(str, pos)
在当前字符串 s 的 pos 索引位置 (默认为 n-1) 开始, 反向查找子串 str
返回找到的位置索引, -1表示查找不到子串 O(n)

```
string s = "0123456789";
cout << (long long)s.rfind("987"); // -1(无符号整数)
cout << s.rfind("789"); // 7
cout << (long long)s.rfind("987", 9); // -1(无符号整数)
cout << s.rfind("789", 9); // 7
```

s.rfind(c, pos)
在当前字符串 s 的 pos 索引位置（默认为 n-1）开始，反向查找字符 c
返回找到的位置索引，-1表示查找不到字符 O(n)

```
string s = "0123456789";
cout << s.rfind('1'); // 1
cout << s.rfind('1', 9); // 1
cout << (long long)s.rfind('a'); // -1(无符号整数)
cout << (long long)s.rfind('a', 9); // -1(无符号整数)
```

14 bitset bool 数组

```
bitset<n> b; (bool 型数组支持位运算)
b.size()
b 中二进制位的个数 O(1)
bitset<10> b;
cout << b.size(); // 10

b.any()
b 中是否存在置为 1 的二进制位，有返回 true，没有返回 false O(n)
bitset<5> b1("10000");
cout << b1.any(); // 1
bitset<5> b2("00000");
cout << b2.any(); // 0

b.none()
b 中是否没有 1，没有返回 true，有返回 false O(n)
bitset<5> b1("10000");
cout << b1.none(); // 0
bitset<5> b2("00000");
cout << b2.none(); // 1

b.count()
返回 b 中为 1 的个数 O(n)
bitset<5> b1("10001");
cout << b1.count(); // 2
bitset<5> b2("00000");
cout << b2.count(); // 0

b.test(pos)
测试 b 在 pos 位置是否为 1，是返回 true O(1)
bitset<5> b("10001");
cout << b.test(0); // 1
cout << b.test(1); // 0

b[pos]
返回 b 在 pos 处的二进制位 O(1)
```

```
bitset<5> b("10001");
cout << b[0]; //1
cout << b[1]; //0

b.set()
把 b 中所有位都置为 1 O(n)
bitset<5> b("00000");
b.set();
//b = "11111"

b.set(pos)
把 b 中 pos 位置置为 1 O(1)
bitset<5> b("10001");
b.set(1);
//b = "11001"

b.reset()
把 b 中所有位都置为 0 O(n)
bitset<5> b("11111");
b.set();
//b = "00000"

b.reset(pos)
把 b 中 pos 位置置为 0 O(1)
bitset<5> b("10001");
b.rset(0);
//b = "00001"

b.flip()
把 b 中所有二进制位取反 O(n)
bitset<5> b("10001");
b.flip();
//b = "01110"

b.flip(pos)
把 b 中 pos 位置取反 O(1)
bitset<5> b("10001");
```

```
b.flip(0);
//b = "00001"
```

15 array 定长数组

```
array<element_type, size> arr;

arr.size()
返回数组 arr 中元素的个数 O(1)
array<int, 3> arr;
cout << arr.size(); // 3

arr.empty()
判断数组 arr 是否为空
空则返回 true，非空则返回 false(返回类型是 bool) O(1)
array<int, 3> arr1;
cout << arr1.empty(); // 1
array<int, 0> arr2;
cout << arr2.empty(); // 0

arr.front()
返回数组 arr 中第一个元素 O(1)
array<int, 3> arr = {1, 2, 3};
cout << arr.front(); // 1

arr.back()
返回数组 arr 中最后一个元素 O(1)
array<int, 3> arr = {1, 2, 3};
cout << arr.back(); // 3

arr.begin()
返回数组 arr 中第一个元素的迭代器 O(1)
array<int, 3> arr = {1, 2, 3};
cout << *arr.begin(); // 1

arr.end()
返回数组 arr 中最后一个元素的后一个位置的迭代器 O(1)

arr.rbegin()
返回数组 arr 中最后一个元素的迭代器 O(1)
```

```cpp
array<int, 3> arr = {1, 2, 3};
cout << *arr.rbegin(); // 3
```

arr.rend()
返回数组 arr 中第一个元素的之前一个位置的迭代器 O(1)

arr.max_size()
返回数组 arr 可容纳元素的最大数量，其值始终等于初始化 array 类的第二个模板参数 size O(1)

```cpp
array<int, 3> arr = {1, 2, 3};
cout << arr.max_size(); // 3
```

arr.fill(x)
将 x 这个值赋值给容器中的每个元素，相当于初始化 O(n)

```cpp
array<int, 3> arr = {1, 2, 3};
arr.fill(0);
//arr = {0, 0, 0}
```

16 常用 STL 函数

```
    accumulate(begin, end, init)
    对一个序列求和初始值为 init
    vector<int> v = {1, 2, 3, 4, 5};
    cout << accumulate(v.begin(), v.end(), 0); // 15
    cout << accumulate(v.begin(), v.begin() + 2, 0); // 3

    atoi(const char *)
    将字符串转换为 int 类型, 不会做范围检查
    string s1 = "1234";
    char s2[10] = "6789";
    cout << atoi(s1.c_str()); // 1234
    cout << atoi(s2); // 6789

    fill(begin, end, x)
    区间 [begin, end) 都赋值为 x
    vector<int> v = {1, 2, 3, 4};
    fill(v.begin(), v.end(), 9);
    for(int i = 0; i < v.size(); i++){
    ^^Icout << v[i] << " ";
^^I}
    // v = {9, 9, 9, 9}

    is_sorted(begin, end)
    判断是否是升序
    是返回 true, 否返回 false
    vector<int> v1 = {1, 2, 3, 4};
    vector<int> v2 = {4, 3, 2, 1};
    cout << is_sorted(v1.begin(), v1.end()); // 1
    cout << is_sorted(v2.begin(), v2.end()); // 0

    lower_bound(a, a + n, x)
    在 a 数组中查找第一个大于等于 x 的元素, 返回该元素的地址
    如果未找到, 返回尾地址的下一个位置的地址
    int a[5] = {1, 2, 2, 4, 5};
    cout << lower_bound(a, a + 5, 2) - a; // 1
```

```
int b[5] = {1, 2, 3, 4, 5};
cout << lower_bound(b, b + 5, 3) - b; // 2
```

upper_bound(a, a + n, x)
在 a 数组中查找第一个大于 x 的元素，返回该元素的地址
如果未找到，返回尾地址的下一个位置的地址

```
int a[5] = {1, 2, 2, 4, 5};
cout << upper_bound(a, a + 5, 2) - a; // 3
int b[5] = {1, 2, 3, 4, 5};
cout << upper_bound(b, b + 5, 3) - b; // 3
```

max_element(begin, end)
返回最大元素的地址

```
int a[5] = {1, 2, 2, 4, 5};
cout << *max_element(a, a + 5); // 5
```

min_element(begin, end)
返回最小元素的地址

```
int a[5] = {1, 2, 2, 4, 5};
cout << *min_element(a, a + 5); // 5
```

max(a, b)
返回最大的那个元素

```
cout << max(2, 3); // 3
cout << max(3, 4); // 4
```

min(a, b)
返回最小的那个元素

```
cout << max(2, 3); // 2
cout << max(3, 4); // 3
```

max({a, b, c, d})
返回最大的那个元素

```
cout << max({3, 5, 4, 7}); // 7
cout << max({3, 9, 4, 7}); // 9
```

min({a, b, c, d})

返回最小的那个元素

```
cout << min({3, 5, 4, 7}); // 3
cout << min({3, 9, 4, 7}); // 3
```

next_permutation(begin, end)

求序列的下一个排列，如果已经是最大就返回 false

```
int a[5] = {1, 2, 3, 4, 5};
do{
    for(int i = 0; i < 5; i++){
        cout << a[i] << " ";
    }
    cout << endl;
}while(next_permutation(a, a + 5));
```

prev_permutation(beg, end)

求序列的上一个排列，如果已经是最小就返回 false

```
int a[5] = {5, 4, 3, 2, 1};
do{
    for(int i = 0; i < 5; i++){
        cout << a[i] << " ";
    }
    cout << endl;
}while(prev_permutation(a, a + 5));
```

reverse(begin, end)

反转 [begin, end)

```
int a[5] = {1, 2, 3, 4, 5};
reverse(a, a + 5);
for(int i = 0; i < 5; i++){
    cout << a[i] << " ";
}
// a = {5, 4, 3, 2, 1}
```

sort(begin, end)

对 [begin, end) 排序

```
int a[5] = {5, 3, 4, 1, 2};
sort(a, a + 5);
```

```cpp
for(int i = 0; i < 5; i++){
    cout << a[i] << " ";
}
// a = {1, 2, 3, 4, 5}
```

stoi(const string*)

将字符串转换为 int 类型

```cpp
string s = "1233";
cout << stoi(s); // 1233
```

to_string(int)

将 int 转换为字符串类型

```cpp
int num = 1233;
cout << to_string(num); // 1233
```

__gcd(a, b)

求 a 和 b 的最大公约数/最大公因数

```cpp
cout << __gcd(12, 16); // 4
cout << __gcd(16, 24); // 8
```

__lg(x)

求以 2 为底 x 的对数

```cpp
cout << __lg(4); // 2
cout << __lg(9); // 3
```

17 C++ 特性

输入/输出
默认支持输入的类型有 char, int, long, long long, string, char *s;
默认支持输出的类型有 char, int, long, long long, string, char *s;
endl 不只有换行功能还可以刷新输出流，所以会导致输出速度变慢

输入方式一

```
int a, b;
cin >> a;
cin >> b;
```

输入方式二

```
int a, b;
cin >> a >> b;
```

输出方式一

```
int a = 10, b = 99;
cout << a;
cout << b;
```

输出方式二

```
int a = 10, b = 99;
cout << a << b;
```

输出定点小数

```
cout << fixed << setprecision(10) << 1 << endl;
//小数点后 10 位

cout << setprecision(10) << 1 << endl;
//一共 10 位
```

简单 lamabada 函数

不能使用外面的变量

```
auto 函数名 = [](参数列表){
    return 返回值;
```

```
};
auto check = [](int x){
    if(x > 10){
        return 1;
    }else{
        return 0;
    }
};
```

能使用外面的变量

```
auto 函数名 = [&](参数列表){
    return 返回值;
};
int target = 10;
auto check = [&](int x){
    if(x > target){
        return 1;
    }else{
        return 0;
    }
}
```

带递归

```
auto dfs = [](auto &&dfs, 参数列表){
    dfs(dfs, 参数列表);
    return 返回值;
}
```

struct 不同之处

1可以在里面写函数，但是外面的函数无法用里面的函数

```
struct Node{
    int x, y;
    void print(){
        cout << x << " " << y << endl;
    }
}
```

2可以重写构造函数，构造函数可以不止一个

```
struct Node{
    int x, y;
    Node(int X, int Y): x(X), y(Y){
    }
}

struct Node{
    int x, y, z;
    Node(int X, int Y){
        this->x = X, this->y = Y, this->z = X * Y;
    }
    Node(int X, int Y, int Z){
        this->x = X, this->y = Y, this->z = Z;
    }
}

struct Node{
    int x, y, z;
    Node(int X, int Y): x(X), y(Y){
        this->z = X * Y;
    }
}
```

3可以重载运算符

```
struct Node{
    int x, y;
    bool operator < (const Node &p)const{
        if(x != p.x){
            return x < p.x;
        }else{
            return y < p.y;
        }
    }
}
```

```
struct Node{
    int x, y;
}
bool operator < (const Node &p1, const Node &p2){
    if(p1.x != p2.x){
        return p1.x < p2.x;
    }else{
        return p1.y < p2.y;
    }
}
```

4可以重载输入输出运算符 (vector, deque 也可以同理重载)

```
struct Node{
    int x, y;
    friend ostream &operator<<(ostream &output, const Node &p){
        cout << p.x << " " << p.x;
        return output;
    }

    friend istream &operator>>(istream &input, Node &p ){
        cin >> p.x >> p.y;
        return input;
    }
}

struct Node{
    int x, y;
}
ostream &operator<<(ostream &output, const Node &p){
    cout << p.x << " " << p.x;
    return output;
}

istream &operator>>(istream &input, Node &p ){
    cin >> p.x >> p.y;
    return input;
}
```

模板 (相当于 T 是一个类型替代符)

```
template <typename T>
T const& Max (T const& a, T const& b) {
    return a < b ? b : a;
}
```

哈希重载

完全版

```
template <typename T>
inline void hash_combine(std::size_t &seed, const T &val) {
    seed ^= std::hash<T>()(val) + 0x9e3779b9 + (seed << 6) + (seed >> 2);
}

template <typename T> inline void hash_val(std::size_t &seed, const T
↪  &val) {
    hash_combine(seed, val);
}
template <typename T, typename... Types>
inline void hash_val(std::size_t &seed, const T &val, const Types &...
↪  args) {
    hash_combine(seed, val);
    hash_val(seed, args...);
}

template <typename... Types>
inline std::size_t hash_val(const Types &... args) {
    std::size_t seed = 0;
    hash_val(seed, args...);
    return seed;
}

struct pair_hash {
    template <class T1, class T2>
    std::size_t operator()(const std::pair<T1, T2> &p) const {
        return hash_val(p.first, p.second);
    }
```

```
};
unordered_map<pair<long long, long long>, long long, pair_hash> mp;
unordered_set<pair<long long, long long>, pair_hash> mp;

pair 版本
struct pair_hash {
    template <class T1, class T2>
    size_t operator()(const pair<T1, T2> &p) const {
        size_t seed = 0;
        seed ^= hash<T1>()(p.first) + 0x9e3779b9 + (seed << 6) + (seed >>
        ↪  2);
        seed ^= hash<T2>()(p.second) + 0x9e3779b9 + (seed << 6) + (seed >>
        ↪  2);
        return seed;
    }
};
unordered_map<pair<long long, long long>, long long, pair_hash> mp;
unordered_set<pair<long long, long long>, pair_hash> mp;
```