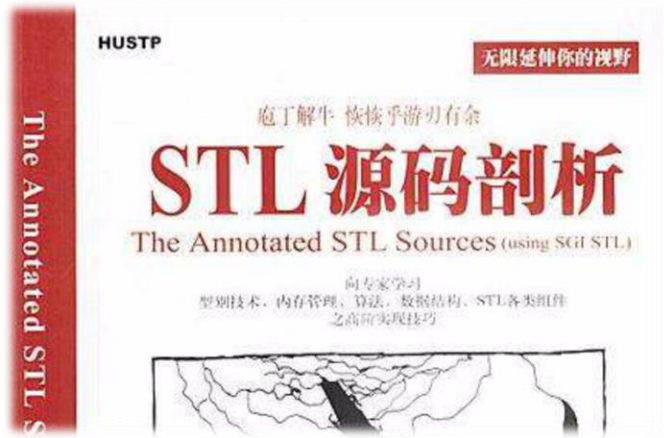
STL容器与算法应用

Cu_OH_2

STL简介

- 全称：标准模板库（由编译器实现于头文件中，命名空间 std）
- 六大组件：**容器/算法**/迭代器/仿函数/配接器/配置器
- 泛型编程：将数据结构、算法和数据类型分离
- 作用：降低代码量/提高可读性



侯捷《STL源码剖析》

STL简介

- 容器：封装了数据结构的模板类
`std::vector/std::deque/std::set/std::map/…`

- 算法：通过迭代器操作数据结构的函数
`std::upper_bound()/std::sort()/std::nth_element()/…`

- 迭代器：容器与算法之间的接口

`std::vector<int>::iterator/std::set<int>::iterator/…`

迭代器 iterator

- 行为类似指针（元素巡访、内容提领、成员访问）
- 规则：**前闭后开** [begin, end)
  - 便于简化循环书写 `for(auto it=begin;it!=end;it++)`
  - 便于统计区间长度 length = end - begin
  - 便于表示区间分割 [begin, mid) + [mid, end)
  - .....
- 每个容器都有自己的专属迭代器类型

```
set<string>::iterator it = st.begin(); // 一般用 auto [C++11]
it++; // 元素巡访
cout << *it << '\n'; // 内容提领
cout << it->size() << '\n'; // 成员访问
```

使用例

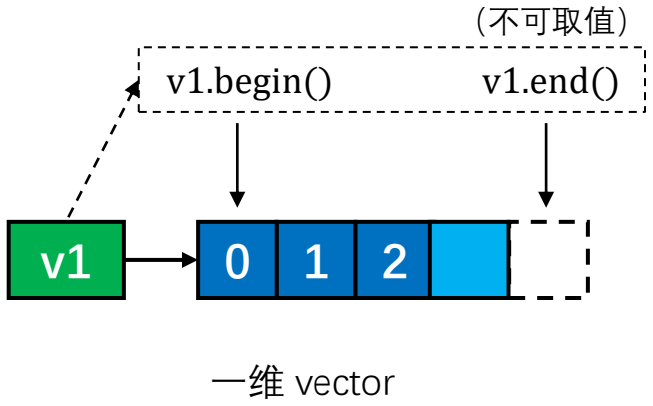
vector

- 线性连续空间（“动态数组”）
- 大小动态变化/支持随机访问
- capacity/size
- 典型应用：邻接表

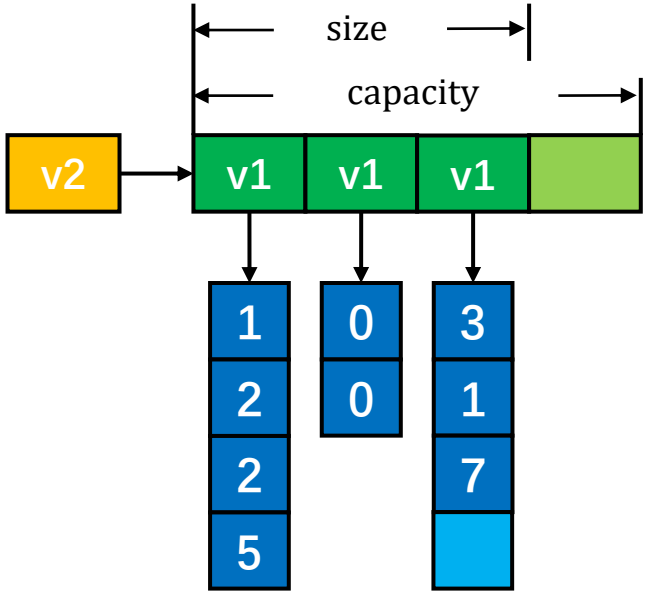
```
vector<int> vec0; // 创建空 vector
vector<int> vec1(100, 6); // 用 6 初始化所有元素
vector<vector<int>> vec2(n, vector<int>(m)); // n*m 的二维 vector

vec1[0] = vec2[x][y]; // 访问和赋值
vec1.push_back(1); // 在尾部插入元素
vec2.pop_back(); // 删除尾部元素
cout << vec1.size() << '\n'; // 大小
vec1.clear(); // 清空
```

使用例



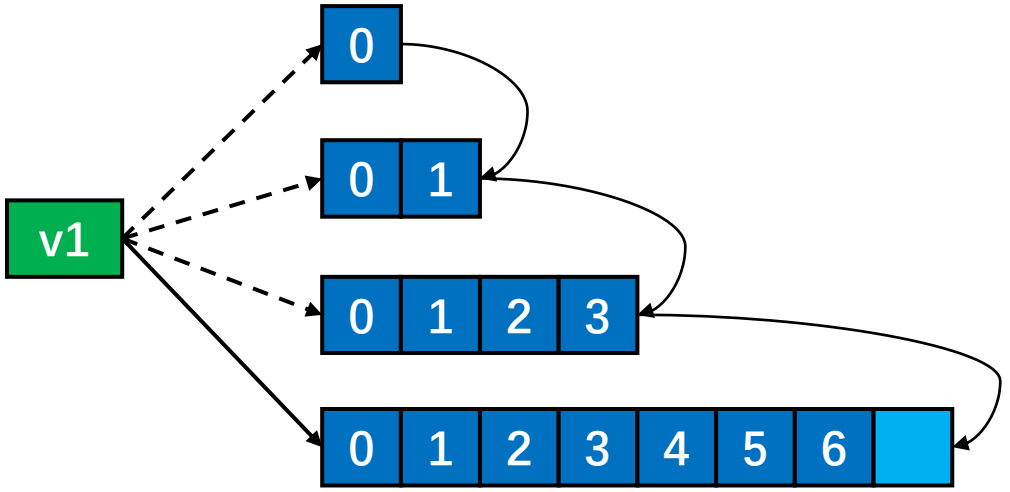
一维 vector



二维 vector

vector

- 常用操作
  - push_back()：尾部插入，均摊 O(1)
  - pop_back()：尾部删除，O(1)
  - operator[]()：随机访问，O(1)
  - insert()/erase()：插入/删除，O(n)
  - clear()：清空，O(n)
  - resize()：更改 size
  - size()：返回 size
  - begin()/end()：返回首尾迭代器
  - ……



假设插入了 n 个元素, 此时 capacity 为 c. 则 $c < 2n$.

总共使用的空间数 $c + \frac{c}{2} + \frac{c}{4} + \frac{c}{8} + \cdots + 1 < 2c < 4n \in O(n)$.

每块空间最多仅在开辟/插入/复制/释放时分别使用一次,

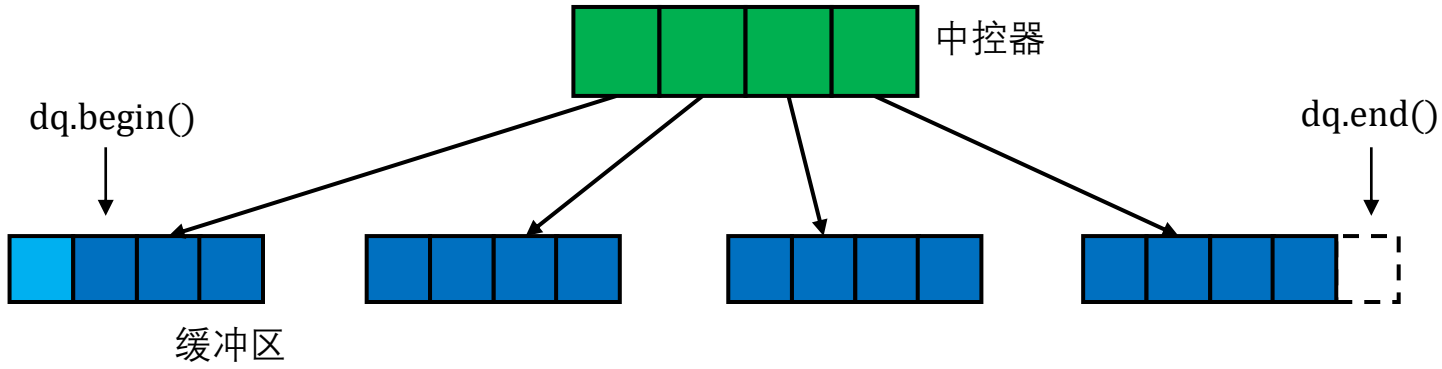
因此总时间 $O(n)$, 均摊 $O(1)$.

deque

- 分段连续空间（双端队列）
- 支持头尾插入删除
- 逻辑上“整体连续”
- 功能自由度高
- 支持下标访问（效率略低）

```cpp
deque<int> dq;
dq.push_front(1);
dq.push_back(2);
dq.pop_front();
cout << dq[0] << '\n';
cout << dq.back() << '\n';
```

使用例



deque 的内部结构

deque

- 常用操作
  - push_front()/push_back()：头/尾插入，均摊 O(1)
  - pop_front()/pop_back()：头/尾删除，O(1)
  - operator[]()：随机访问，O(1)
  - clear()：清空，O(n)
  - front()/back()：返回头部/尾部元素
  - size()：返回元素个数
  - ……

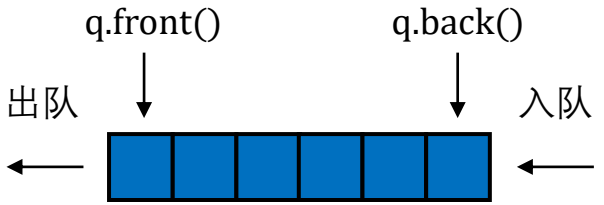
```
using ll = long long;
struct Node
{
    ll key, id;
};
```

```
vector<deque<Node>> dq(w[i]);
auto key = [&](int j) { return dp[j] - j / w[i] * v[i]; };
auto join = [&](int j)
{
    auto& q = dq[j % w[i]];
    while (q.size() && key(j) >= q.back().key) q.pop_back();
    q.push_back({ key(j),j });
    return;
};
for (int j = m; j >= max(0ll, m - k[i] * w[i]); --j) join(j);
for (int j = m; j >= w[i]; --j)
{
    auto& q = dq[j % w[i]];
    while (q.size() && q.front().id >= j) q.pop_front();
    if (j - k[i] * w[i] >= 0) join(j - k[i] * w[i]);
    dp[j] = max(dp[j], q.front().key + j / w[i] * v[i]);
}
```

单调队列优化多重背包问题片段

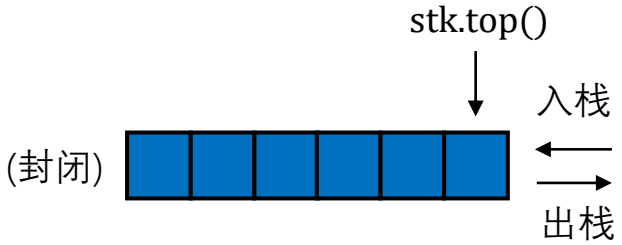
queue/stack

- 基于 deque 的容器**配接器**
- 功能是 deque 的子集
- 限制了下标访问和部分头尾操作
- 不允许遍历，没有迭代器



queue 的逻辑结构

- queue：先进先出的线性结构（队列）
- stack：后进先出的线性结构（栈）



stack 的逻辑结构

queue

- 常用操作
  - push()：尾部插入
  - pop()：头部删除
  - front()/back()：返回头部/尾部元素
  - size()：返回元素个数
  - ……
- 典型应用：BFS

stack

- 常用操作
  - push()：尾部插入
  - pop()：尾部删除
  - top()：返回尾部元素
  - size()：返回元素个数
  - ……
- 典型应用：单调栈

priority_queue

- 二叉堆实现的优先队列
- 对于原生数据类型，默认大者优先
- 支持自定义比较方法
- 常用操作
  - push()：插入，O(logn)
  - pop()：删除，O(logn)
  - top()：返回堆顶（最值）
  - size()：返回元素个数
  - ……
- 典型应用：优化 Dijkstra 算法

```
struct T
{
    int x;
    bool operator<(const T& rhs) const { return x < rhs.x; }
};
```

重载比较运算符 operator<

```
auto cmp1 = [&](const T& lhs, const T& rhs) { return lhs.x < rhs.x; };
```

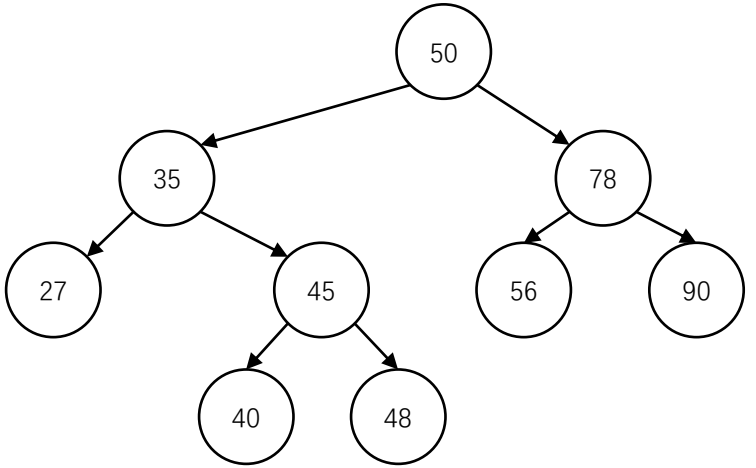
```
bool cmp2(const T& lhs, const T& rhs) { return lhs.x < rhs.x; }
```

```
priority_queue<int> pq1; // 大数优先（大根堆）
priority_queue<int, vector<int>, greater<int>> pq2; // 小数优先（小根堆）
priority_queue<T, vector<T>, decltype(cmp1)> pq3(cmp1); // lambda函数
priority_queue<T, vector<T>, decltype(&cmp2)> pq4(cmp2); // 普通函数
```

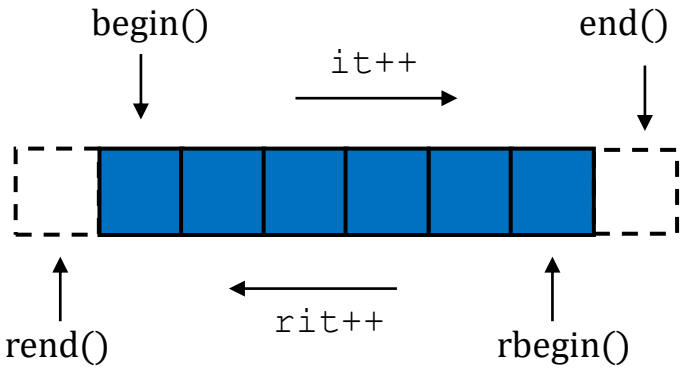
不同比较方式的 priority_queue 对象的定义
（decltype为类型推导[C++11]）

set/map

- 红黑树（有序集合/有序映射）
- 键值 key（不可修改）/实值 value（可修改）
- 键值必须可比较
- 不存在两个键值相等的结点
- 支持自定义比较方法
- 迭代器分为正向和反向
- multiset/multimap



二叉搜索树



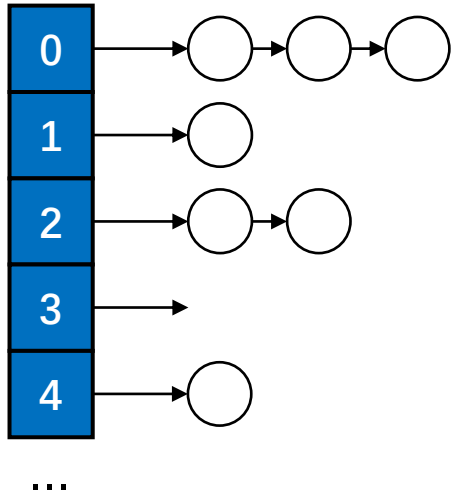
逻辑结构

set/multiset/map

- 常用操作
  - insert()：插入结点（自动去重），O(logn)
  - erase()：根据值或结点删除结点，O(logn)
  - count()：根据值统计结点个数，O(logn)
  - find()：搜索结点，返回迭代器，O(logn)
  - clear()：清除所有结点
  - size()：返回结点个数
  - begin()/end()/rbegin()/rend()：返回正反首尾迭代器
  - lower_bound()/upper_bound()：二分查找，O(logn)
  - operator[]()（map）：取键值对应的实值（如果不存在将会创建结点），O(logn)

unordered_set/unordered_map [C++11]

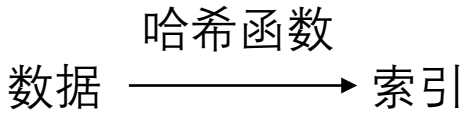
- 哈希表（无序集合/无序映射）
- 哈希函数
- 生日悖论与哈希冲突
- 解决冲突：开链法
- 减少冲突：预设质数作为表大小



开链法哈希表

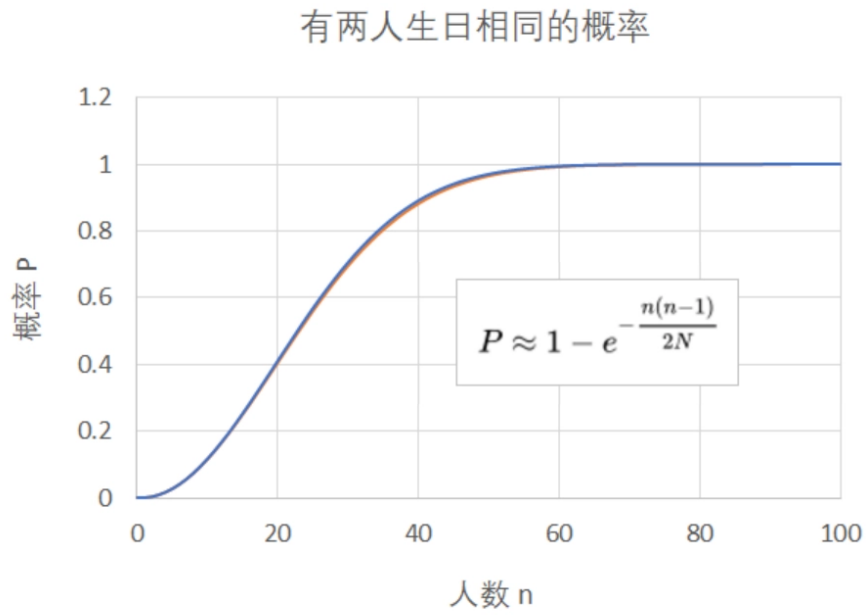
```
struct CustomHash
{
    static uint64_t splitmix64(uint64_t x)
    {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const
    {
        static const uint64_t FIXED_RANDOM = chrono::steady_clock::now().time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};
```



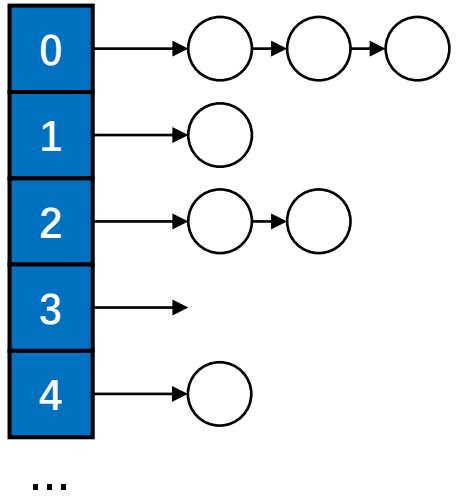
生日悖论

- 在不少于 23 个人中至少有两人生日相同的概率大于 50%
- 大小为 N 的哈希表，填入 n 个数，碰撞概率为 $1-e^{-\frac{n(n-1)}{2N}}$
- 通常在 $n \in \Theta(\sqrt{N})$ 时，哈希冲突带来的常数不可忽略



unordered_set/unordered_map [C++11]

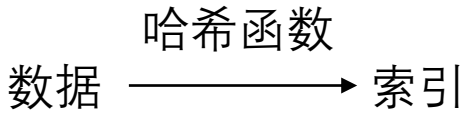
- 哈希表（无序集合/无序映射）
- 哈希函数
- 生日悖论与哈希冲突
- 解决冲突：开链法
- 减少冲突：预设质数作为表大小



开链法哈希表

```
struct CustomHash
{
    static uint64_t splitmix64(uint64_t x)
    {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const
    {
        static const uint64_t FIXED_RANDOM = chrono::steady_clock::now().time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};
```



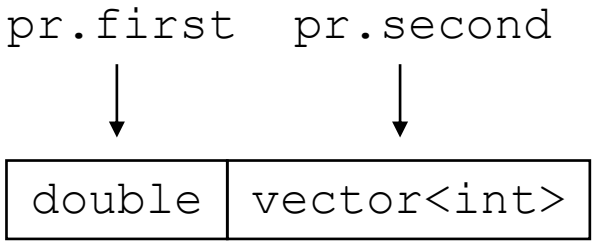
unordered_set/unordered_map

- 常用操作
  - insert()：插入结点（自动去重），O(1)
  - erase()：根据值或结点删除结点，O(1)
  - count()：根据值统计结点个数，O(1)
  - find()：搜索结点，返回迭代器，O(1)
  - clear()：清除所有结点
  - size()：返回结点个数
  - operator[]()（unordered_map）：取键值对应的实值（如果不存在将会创建结点），O(1)

注意：常数过大时有可能比 O(log) 还慢!

pair

- 任意两种数据类型的组合
- 自带比较函数
- make_pair()
- map 中的每个结点都是 pair



pair<double, vector<int>>

```
pair<int, int> pr = { 1, 2 };
// pair<int, int> pr = make_pair(1, 2);
map<string, int> mp;
for (auto pr : mp)
{
    cout << pr.first << '\n'; // string
    cout << pr.second << '\n'; // int
}
```

使用例

sort()

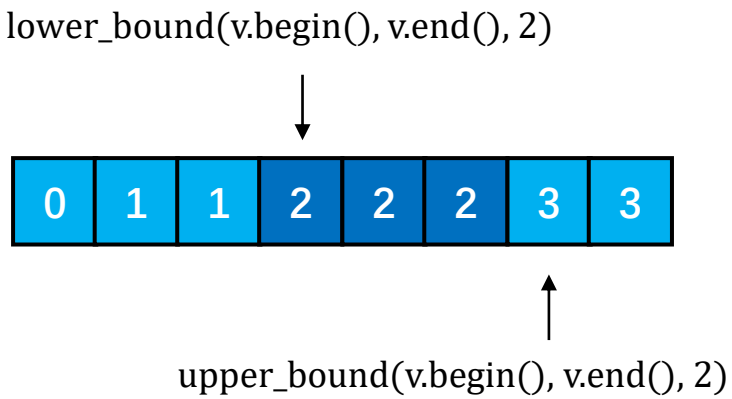
- 排序策略
  - 数据量大时采用**快速排序**
  - 分段后对较小段采用**插入排序**
  - 递归层次过深时采用**堆排序**
- 效率较高：时间复杂度 O(nlogn)
- 默认升序
- 支持自定义比较方法
  - 重载比较运算符
  - 传入函数/仿函数/lambda函数

```
int a[4] = { 2, 1, 3, 4 };
sort(a, a + 4);
sort(a + 1, a + 3, greater<int>());
vector<int> v = { 8, 6, 4 };
sort(v.begin(), v.end());
sort(v.begin(), v.end(), [](int x, int y) { return (x & 1) > (y & 1); });
```

使用例

lower_bound()/upper_bound()

- 二分序列查找元素，O(logn)
- 前提：序列有序
- “下界”：第一个**大于等于**指定值的位置
- “上界”：第一个**大于**指定值的位置
- 符合“**前闭后开**”原则



其他

- max_element()/min_element()
- fill()
- reverse()
- min()/max()

```
vector<int> v = { 3, 1, 4, 1, 5, 9, 2, 6 };
*max_element(v.begin(), v.end()); // 9
*min_element(v.begin(), v.end()); // 1
count(v.begin(), v.end(), 1); // 2
fill(v.begin(), v.end(), 0);
reverse(v.begin(), v.end());
min(1, 2); // 1
max({ 6, 5, 4, 7, 2 }); // 7
```

例题

· priority_queue/multiset

2558. 从数量最多的堆取走礼物 - 力扣（LeetCode）

· unordered_map

1. 两数之和 - 力扣（LeetCode）

· sort()

1465. 切割后面积最大的蛋糕 - 力扣（LeetCode）

· lower_bound()/upper_bound()

34. 在排序数组中查找元素的第一个和最后一个位置 - 力扣（LeetCode）

谢谢观看

Cu_OH_2