第八章 C++ STL

STL 是提高 C++编写效率的一个利器。

——闫学灿

1. #include <vector>

vector 是变长数组，支持随机访问，不支持在任意位置 O(1)插入。为了保证效率，元素的增删一般应该在末尾进行。

声明

```
#include <vector>      头文件
vector<int> a;         相当于一个长度动态变化的 int 数组
vector<int> b[233];    相当于第一维长 233，第二位长度动态变化的 int 数组
struct rec{…};
vector<rec> c;         自定义的结构体类型也可以保存在 vector 中
```

size/empty

size 函数返回 vector 的实际长度（包含的元素个数），empty 函数返回一个 bool 类型，表明 vector 是否为空。二者的时间复杂度都是 O(1)。

所有的 STL 容器都支持这两个方法，含义也相同，之后我们就不再重复给出。

clear

clear 函数把 vector 清空。

迭代器

迭代器就像 STL 容器的“指针”，可以用星号“*”操作符解除引用。

一个保存 int 的 vector 的迭代器声明方法为：

```
vector<int>::iterator it;
```

vector 的迭代器是“随机访问迭代器”，可以把 vector 的迭代器与一个整数相加减，其行为和指针的移动类似。可以把 vector 的两个迭代器相减，其结果也和指针相减类似，得到两个迭代器对应下标之间的距离。

begin/end

begin 函数返回指向 vector 中第一个元素的迭代器。例如 a 是一个非空的 vector，则*a.begin()与 a[0]的作用相同。

所有的容器都可以视作一个“前闭后开”的结构，end 函数返回 vector 的尾部，即第 n 个元素再往后的“边界”。*a.end()与 a[n]都是越界访问，其中 n=a.size()。

下面两份代码都遍历了 vector<int>a，并输出它的所有元素。

```
for (int I = 0; I < a.size(); I ++) cout << a[i] << endl;
for (vector<int>::iterator it = a.begin(); it != a.end(); it ++) cout << *it << endl;
```

front/back

front 函数返回 vector 的第一个元素，等价于*a.begin() 和 a[0]。

back 函数返回 vector 的最后一个元素，等价于*==a.end() 和 a[a.size() − 1]。

push_back() 和 pop_back()

a.push_back(x) 把元素 x 插入到 vector a 的尾部。

b.pop_back() 删除 vector a 的最后一个元素。

2. #include <queue>

头文件 queue 主要包括循环队列 queue 和优先队列 priority_queue 两个容器。

声明

```
queue<int> q;
struct rec{…}; queue<rec> q;  //结构体 rec 中必须定义小于号
priority_queue<int> q;      // 大根堆
priority_queue<int, vector<int>, greater<int> q;  // 小根堆
priority_queue<pair<int, int>>q;
```

循环队列 queue

push 从队尾插入

pop 从队头弹出

front 返回队头元素

back 返回队尾元素

优先队列 priority_queue

push 把元素插入堆

pop 删除堆顶元素

top 查询堆顶元素（最大值）

3. #include <stack>

头文件 stack 包含栈。声明和前面的容器类似。

push 向栈顶插入

pop 弹出栈顶元素

4. #include <deque>

双端队列 deque 是一个支持在两端高效插入或删除元素的连续线性存储空间。它就像是 vector 和 queue 的结合。与 vector 相比，deque 在头部增删元素仅需要 O(1)的时间；与 queue 相比，deque 像数组一样支持随机访问。

[] 随机访问

begin/end，返回 deque 的头/尾迭代器

front/back 队头/队尾元素

push_back 从队尾入队

push_front 从队头入队

pop_back 从队尾出队

pop_front 从队头出队

clear 清空队列

5. #include <set>

头文件 set 主要包括 set 和 multiset 两个容器，分别是“有序集合”和“有序多重集合”，即前者的元素不能重复，而后者可以包含若干个相等的元素。set 和 multiset 的内部实现是一棵红黑树，它们支持的函数基本相同。

声明

set<int> s;

struct rec{…}; set<rec> s; // 结构体 rec 中必须定义小于号

multiset<double> s;

size/empty/clear

与 vector 类似

迭代器

set 和 multiset 的迭代器称为“双向访问迭代器”，不支持“随机访问”，支持星号(*)解除引用，仅支持”++”和––“两个与算术相关的操作。

设 it 是一个迭代器，例如 set<int>::iterator it;

若把 it++，则 it 会指向“下一个”元素。这里的“下一个”元素是指在元素从小到大排序的结果中，排在 it 下一名的元素。同理，若把 it––，则 it 将会指向排在“上一个”的元素。

begin/end

返回集合的首、尾迭代器，时间复杂度均为 O(1)。

s.begin() 是指向集合中最小元素的迭代器。

s.end() 是指向集合中最大元素的下一个位置的迭代器。换言之，就像 vector 一样，是一个“前闭后开”的形式。因此––s.end()是指向集合中最大元素的迭代器。

insert

s.insert(x)把一个元素 x 插入到集合 s 中，时间复杂度为 O(logn)。

在 set 中，若元素已存在，则不会重复插入该元素，对集合的状态无影响。

find

s.find(x) 在集合 s 中查找等于 x 的元素，并返回指向该元素的迭代器。若不存在，则返回 s.end()。时间复杂度为 O(logn)。

lower_bound/upper_bound

这两个函数的用法与 find 类似，但查找的条件略有不同，时间复杂度为 O(logn)。

s.lower_bound(x) 查找大于等于 x 的元素中最小的一个，并返回指向该元素的迭代器。

s.upper_bound(x) 查找大于 x 的元素中最小的一个，并返回指向该元素的迭代器。

erase

设 it 是一个迭代器，s.erase(it) 从 s 中删除迭代器 it 指向的元素，时间复杂度为 O(logn)

设 x 是一个元素，s.erase(x) 从 s 中删除所有等于 x 的元素，时间复杂度为

O(k+logn)，其中 k 是被删除的元素个数。

count

s.count(x) 返回集合 s 中等于 x 的元素个数，时间复杂度为 O(k +logn)，其中 k 为元素 x 的个数。

6. #include <map>

map 容器是一个键值对 key–value 的映射，其内部实现是一棵以 key 为关键码的红黑树。Map 的 key 和 value 可以是任意类型，其中 key 必须定义小于号运算符。

声明

map<key_type, value_type> name;

例如：

map<long, long, bool> vis;

map<string, int> hash;

map<pair<int, int>, vector<int>> test;

size/empty/clear/begin/end 均与 set 类似。

Insert/erase

与 set 类似，但其参数均是 pair<key_type, value_type>。

find

h.find(x) 在变量名为 h 的 map 中查找 key 为 x 的二元组。

[]操作符

h[key] 返回 key 映射的 value 的引用，时间复杂度为 O(logn)。

[]操作符是 map 最吸引人的地方。我们可以很方便地通过 h[key]来得到 key 对应的 value，还可以对 h[key]进行赋值操作，改变 key 对应的 value。