程序设计思维与实践

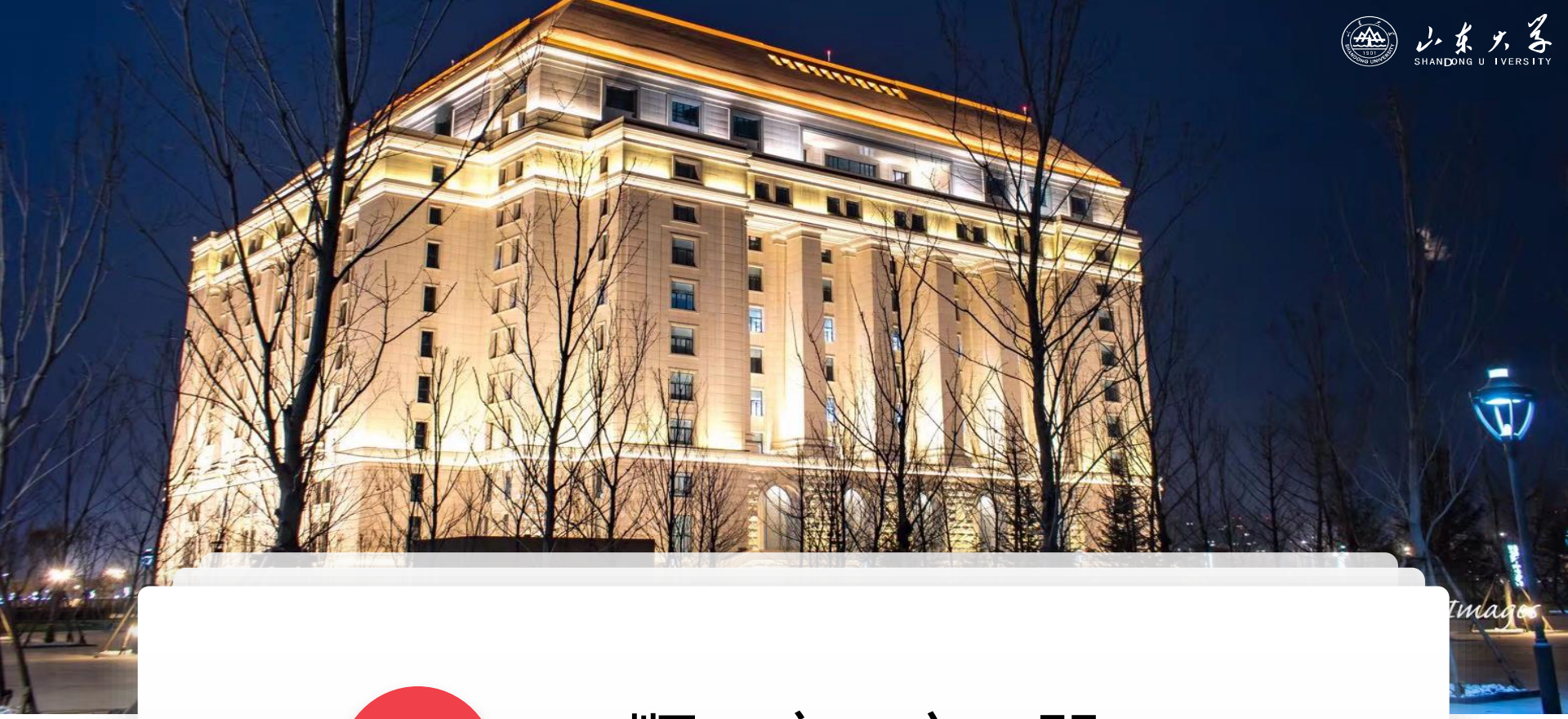
Thinking and Practice in Programming

C++ 与 STL | 内容负责：苗顺源/尹浩飞

知识梳理

- 关于C++的STL，我们学习了哪些
- 想了解更多知识，可以查阅：C++ 参考手册(中文) 或 C++官方文档(英文)

算法	<algorithm>库	sort/next_permutation/二分 等
顺序容器	vector	向量（不定长数组）
	list	链表
	string	字符串
	deque	双端队列
关联容器	map/multimap	映射/可重映射
	set/multiset	集合/可重集合
	unordered_map	基于Hash的映射（了解，C++11）
	unordered_set	基于Hash的集合（了解，C++11）
容器适配器	stack	栈
	queue	队列
	priority_queue	优先队列（堆）



1

顺 序 容 器

Sequence container

pair

- pair（元组）可以存储两个不同类型的数据，一定程度上可以取代自定义结构体。
- pair内置了各种比较函数，逻辑是优先比较first的大小，如果first的大小相同再比较second的大小。

```
/* 声明 */
pair<int, double> p;

/* 赋值 */
p.first = 0; p.second = 1.1;
p = make_pair(0, 1.1);
// 上面两行是等价的

/* 比较 */
pair<int, double> a(1,1.5), b(1,3.0), c(0,3.1);
cout << (a<b) << endl; // 输出1
cout << (a<c) << endl; // 输出0
```

vector

- vector 是不定长数组容器。其数组长度的扩增策略是每次乘 2，所以倘若数组长度下界较为确定的话，声明 vector 时传入初始大小是比较明智的。
- 对于容器类，较为关注的是它的声明、赋值、遍历、清空操作及相应复杂度

```
/* 声明 */
vector<int> v(10);     // 初始大小为10
vector<int> v(10, 0); // 初始大小为10，每个值都是0
vector<vector<int> > G(n+1,vector<int>(m+1));  // 二维动态数组声明
/* 赋值 */
v.push_back(20); // 末尾放入一个元素20
v[0] = 12; // 首元素改为12
v.insert(v.begin()+1, 1);  // 第2个位置插入1
v.emplace(v.begin()+1, 1); // 第2个位置插入1，不复制构造而直接插入，优化
```

vector

- vector 是不定长数组容器。其数组长度的扩增策略是每次乘 2，所以倘若数组长度下界较为确定的话，声明 vector 时传入初始大小是比较明智的。
- 对于容器类，较为关注的是它的声明、赋值、遍历、清空操作及相应复杂度

```
/* 遍历 */
for(int i=0, n=v.size();i<n;i++) printf("%d\n", v[i]); // 原生
for(vector<int>::iterator it=v.begin();it!=v.end();++it) printf("%d\n", *it); // 迭代器
for(int i : v) printf("%d", i); // for range
for(auto i : v) printf("%d", i); // for range
for(auto &i : v) i++; // 每个数自增1
/* 清空 */
v.clear(); // 整个清空
v.pop_back(); // 删除最后元素
v.erase(v.begin()); // 删除首个元素
v.erase(v.begin(), v.begin()+3); // 删除前3个元素
```

list

- list 是链表容器。将数组换作链表使用，自然更在意的是其增删操作的时间复杂度和存储的空间复杂度。
- 在链表容器中，除了对基本函数了解外，理应对迭代器有较好的理解和应用。若对时空复杂度有更高要求，单向链表 forward_list 也许更适合。

```
/* 声明 */
list<int> l1;
list<int> l2(l1.begin(), l1.end()-1); // 拷贝其他链表的一部分，复杂度是O(n)

/* 赋值 */
l1.push_back(1);  // 在链表尾加一个元素
l1.push_front(1); // 在链表头加一个元素

/* 遍历 */
l1.front(); // 首元素
l1.back();  // 尾元素
for(list<int>::iterator it = l1.begin(); it != l1.end(); it++) printf("%d\n", *it);
for(list<int>::reverse_iterator it = l1.rbegin(); it != l1.rend(); it++) printf("%d\n", *it);
for(auto &i : l) printf("%d ", i);
```

list

- list 是链表容器。将数组换作链表使用，自然更在意的是其增删操作的时间复杂度和存储的空间复杂度。
- 在链表容器中，除了对基本函数了解外，理应对迭代器有较好的理解和应用。若对时空复杂度有更高要求，单向链表 forward_list 也许更适合。

```
/* 迭代器删减 */
list<int>::iterator it = l1.begin();
it++; // 迭代器移动
l1.insert(it, 1);  // 在迭代器后加一个元素
it = l1.erase(it); // 删除迭代器当前元素，必须重新赋值迭代器否则失效，赋值后迭代器等于被删元素下一个

/* 清空 */
l1.pop_front(); // 删除首元素
l1.pop_back();  // 删除尾元素
l1.clear();     // 清空链表

/* 排序，复杂度nlogn */
bool cmp(const int &o1, const int &o2) {return o1 > o2;}
l1.sort(cmp); // 空参数为递减
```

list

- list 是链表容器。将数组换作链表使用，自然更在意的是其增删操作的时间复杂度和存储的空间复杂度。
- 在链表容器中，除了对基本函数了解外，理应对迭代器有较好的理解和应用。若对时空复杂度有更高要求，单向链表 forward_list 也许更适合。

```
/* 链表反向 */
l1.reverse();

/* 删除链表中全部某值元素 */
l1.remove(89);                                                    // 第一种
bool cmp(const int &o) {return o==89;}                l1.remove_if(cmp); // 第二种

/* 去重 */
l1.sort(); l1.unique();
```

deque

- deque与vector类似，它支持vector的所有函数（包括下标访问），并且实现了O(1)复杂度的前端插入与删除。
- 虽说在功能上可以替代vector，但是对比效率还是vector更优秀一些。

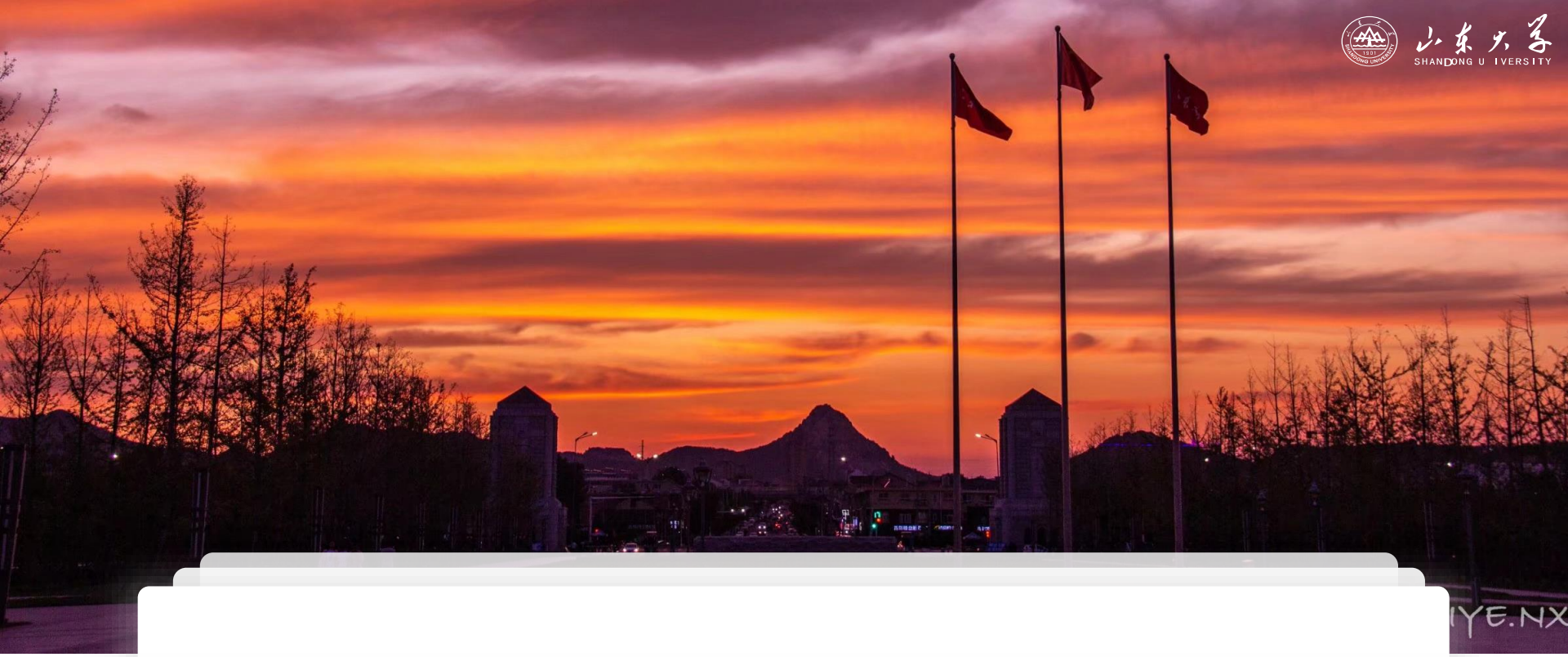
```
/* 声明 */
deque<int> d(10);

/* 赋值 */
d.push_front(13); // 添加至开头
d.push_back(25); // 添加至末尾
d[0] = 64;

/* 删除 */
d.pop_front(); // 删除首个元素(O(1))
d.pop_back(); // 删除尾部元素(O(1))
```

string

- 在C语言中，字符串就是字符数组，而不是像int/double那样的“一等公民”，使用起来处处受限
- C++提供了#include <string>中的string类型，重载了很多运算符，使程序更加自然，简单。

```
string s;
cin >> s;   //字符串的声明 输入 输出
cout << s;

cout << s.size() << endl; //字符串长度

string a;

s += a;     //字符串连接字符串
s += '+';   //字符串连接字符

if (s == a)cout << "true" << endl; //判断两个串是否完全相等
if (s < a)cout << "true" << endl;  //判断两个串的字典序大小

for (int i = 0; i < s.size(); ++i)
    cout << s[i];   //遍历字符串的所有字符
```

```
s.substr(2,3); // 取下标从2开始，长度为3的字串
s.push_back('1'); // 在末尾添加一个字符
s.c_str(); // 返回一个常量字符数组的指针
```



2 容器适配器&算法

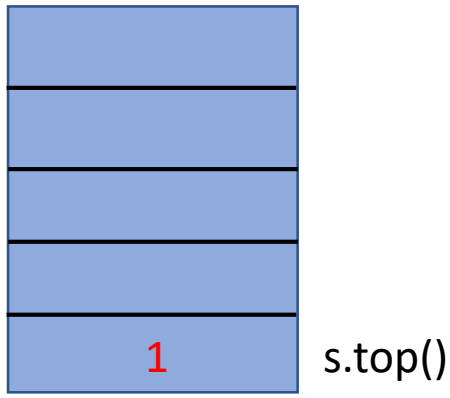
Container adaptors & Algorithms

stack

- 栈，先进后出（后进先出）的数据结构
- 概念数据结构上都有所涉及，这里主要关注 C++ 标准函数库的使用

  “简单过一下”

```
/* 声明 */
stack<int> s;
stack<int, vector<int> > s; // 指定底层容器的栈
stack<int, list<int> > s;   // 指定底层容器的栈

/* 赋值 */
s.push(1); // 将1压栈

/* 访问 */
s.top();   // 访问栈顶

/* 清空 */
s.pop();                            // 弹出栈顶
while(!s.empty()) s.pop();          // 清空栈
for(int i=s.size(); i; i--) s.pop(); // 清空栈
```



queue & priority_queue

- 队列，先进先出；优先队列，又称为“堆”

```
/* 声明 */
queue<int> q;

priority_queue<int> pq;                                  // 优先队列，大根堆

// 大根堆 O(n) 线性构造
int a[] = {1,3,4,6,78,9}
priority_queue<int> pq(a, a+6);

priority_queue<int, vector<int>, greater<int> > pq; // 小根堆，结构体重载>方法
priority_queue<int, vector<int>, less<int> > pq;    // 大根堆，结构体重载<方法

/* 赋值 */
q.push(1); // 将1入队
pq.push(1);

/* 队列访问 */
q.front(); // 访问队首
q.back();  // 访问队尾
```

```
/* 优先队列访问 */
pq.top();  // 访问堆顶

/* 清空，二者是相同的 */
q.pop();                                  // 队首出队
pq.pop();                                 // 弹出堆顶
while(!q.empty()) q.pop();                // 清空队列
for(int i=q.size();i;i--) q.pop(); // 清空队列
```

queue & priority_queue

- 优先队列怎么知道元素的大小判断方法？刚刚的例子因为使用了基本的数据类型，它们自带天然的大小判断方法。如果我们自己实现的复杂数据结构，则需要重写比较方法！
- 结构体-优先队列 写法？ —— 复习结构体比较方法重写

```
struct P {
    int x, y, z;
    bool operator<(const P &p)const {
        if (x != p.x) return x < p.x; // 第一关键字升序
        if (y != p.y) return y > p.y; // 第二关键字降序
        return z < p.z;               // 第三关键字升序
    }
}Ps[1005];

priority_queue<P> heap;               // 大根堆

heap.push({1, 2, 3});
heap.push({2, 2, 3});
heap.push({1, 3, 4});
printf("%d\n", heap.top().x);         // 输出什么？  1 还是 2?
```

补充

- 结构体重载为什么只需要重载一个<号?

- 因为一个<号可以生成所有比较函数

```
bool operator > (const node_time &b)const{return b<*this;}
bool operator <= (const node_time &b)const{return !(b<*this);}
bool operator >= (const node_time &b)const{return !(*this<b);}
bool operator == (const node_time &b)const{return !(b<*this||*this<b);}
bool operator != (const node_time &b)const{return b<*this||*this<b;}
```

algorithm

- 算法库，有 C++ 内置的各种常用算法，常用的有：二分、排序、去重、全排列等，这里关注排序 sort

```
int a[] = {5,3,1,4};
vector<int> b = {5,3,1,4};
sort(a, a+4);                    // 左闭右开区间 升序排序
sort(b.begin()+1, b.end());      // 只排序第2个元素及其后面的元素，升序排序
sort(a, a+4, greater<int>()); // 降序排序
```

- 传参比较方法

```
struct P { int a,b,c; }Ps[10];
bool cmp(const P &p1, const P &p2) { return a!=p.a ? a<p.a : (b!=p.b ? b<p.b : c<p.c); }
sort(Ps, Ps+10, cmp); // 传参比较方法，三关键字升序排序
```

- 结构体重写比较方法

```
struct P {
    int a,b,c;
    bool operator<(const P &p) const {// 第1关键字升序，第2关键字降序，第3关键字升序的多关键字排序
        if(a != p.a) return a < p.a;
        if(b != p.b) return b > p.b;
        return c < p.c;
    }
}Ps[10];
sort(Ps, Ps+10);      // 重载比较方法，第1关键字升序，第2关键字降序，第3关键字升序的多关键字排序
```

algorithm

- 其他常用的函数

```
int a[] = {1, 2, 3, 5, 6, 8};
vector<int> v = {1, 2, 3, 5, 6, 8};
// 传入起始和结束指针/迭代器，返回找到位置的指针/迭代器
cout << *lower_bound(a, a+6, 4) << endl; // 大于等于4的第一个位置
cout << *lower_bound(v.begin(), v.end(), 3) << endl;
cout << *upper_bound(a, a+6, 3) << endl; // 严格大于3的第一个位置
```

```
int a[] = {0, 2, 6, 4, 2, 3, 9, 4, 3, 5};
cout << *min_element(a, a + 10) << endl; // 返回最小值位置
cout << *max_element(a, a + 10) << endl; // 返回最大值位置
pair<int *, int *> minmax = minmax_element(a, a + 10); //同时返回二者
cout << *minmax.first << ' ' << *minmax.second << endl;
```

```
int a[] = {1, 1, 2, 2, 3, 5, 1, 1, 4, 2};
reverse(a+2, a+6); // a = {1, 1, 5, 3, 2, 2, 1, 1, 4, 2}
int *b = unique(a, a+10); // b-a = 7
sort(a, b);
b = unique(a, b); // b-a = 5
do {
    /* 对排列进行操作 */
} while (next_permutation(a, a+5)); // 生成下一个排列
```

Lambda与匿名函数（了解）

- Lambda 不是 C++ 中某个库，而是 C++11 新支持的一个特性，Lambda 是基于数学中的 λ 演算得名的，在 C++11 里表现为匿名函数的支持，可以替代掉一些一次性的谓词函数，起到简化逻辑、增加可读性的作用。
- 是一种 “语法糖” 。
- 这里属于拓展，主要关注怎样用在 sort 的比较函数上。

```
/* 传统的排序 */
bool cmp(int a, int b) {return a<b;}
int a[] = {1,5,3,2};
sort(a, a+4, cmp);

/* 引入Lambda，写匿名函数 */
sort(a, a+4, [](int a,int b)->bool{return a<b;} );
// Lambda 支持的结构体多关键字排序
struct P { int a,b,c; }Ps[10];
sort(Ps, Ps+10, [](P &p1, P &p2)->bool{
    if(p1.a != p2.a) return p1.a < p2.a;
    if(p1.b != p2.b) return p1.b > p2.b;
    return p1.c < p2.c;
});
```

3 关 联 容 器

Associative containers

map

下面先关注用法，再了解原理

- 概念：<key, value> 键值对。给一个 key，可以得到唯一 value
- 基本使用：

```
/* 声明 */
map<int, bool> mp;

/* 赋值 */
mp[13] = true;

/* 查key是否有对应value */
if(mp[13]) printf("visited"); // 这里可以直接如此，而如果 value 类型非 bool 时需换成以下这种
if(mp.find(13) != mp.end()) printf("visited");
if(mp.count(13)) printf("visited");

/* 遍历 */
for(map<int,int>::iterator it=mp.begin();it!=mp.end();it++)
    printf("%d %d\n", it->first, it->second); // 输出是升序的
for(map<int,int>::reverse_iterator it=mp.rbegin();it!=mp.rend();it++)
    printf("%d %d\n", it->first, it->second); // 输出是降序的

/* 遍历 (C++11 特性) */
for(auto &entry : mp) printf("%d %d\n", entry.first, entry.second); // for range
```

map

下面先关注用法，再了解原理

- 概念：<key, value> 键值对。给一个 key，可以得到唯一 value
- 基本使用：

```
/* 清空 */
mp.erase(13); // 以键为关键字删除某该键-值对，复杂度是 log
mp.clear();

/* map声明，而不是声明一个空的map随后在main中赋值 */
map<char, char> mp({
    {'R', 'P'},
    {'P', 'S'},
    {'S', 'R'}
});
```

```
map<string,int> M;
string s1,s2,s3,s4,s5,s6;
int main()
{
    M["one"] = 1;
    M["two"] = 2;
    M["three"] = 3;
    M["four"] = 4;
    M["five"] = 5;
    M["six"] = 6;
    M["seven"] = 7;
    M["eight"] = 8;
    M["nine"] = 9;
    M["zero"] = 0;
```

```
/* 结构体使用map需要重写比较方法 */
struct Point {
    int x, y;
    bool operator<(const Point &p) const {
        return  x!=p.x ? x<p.x : y<p.y;
    }
};
int main()
{
    map<Point, bool> mp;
    printf("%d\n", mp[{1,5}]);
    mp[{1,5}] = true;
    printf("%d\n", mp[{1,5}]);
    return 0;
}
```

set

- 概念：可以理解为数学意义上的“集合”， 三大特性？ （确定/互异/无序）
- 对于一个元素，可以放入集合中；也可查找集合中是否有该元素（或删）

```
/* 声明 */
set<int> s;

/* 赋值 */
s.insert(9);   s.insert(4);   s.insert(6);   s.insert(4);  // set 中有3个元素

/* 遍历 */
if(s.find(1) != s.end()) printf("The element is in set");
if(s.count(1)) printf("The element is in set");
for(set<int>::iterator it=s.begin();it!=s.end();it++) printf("%d", *it); // 全部遍历
for(auto &x : s) printf("%d\n", x);
int sz = s.size(); // 大小

/* 清空 */
s.erase(val); // 删除复杂度是logn
s.clear();

/* 二分 */
set<int>::iterator it = s.lower_bound(5); // 查找第一个大于等于 5 的元素，返回指针
if (it != set.end())    // 存在该元素
    printf("%d\n", *it);// 输出什么?
```

multimap/multiset

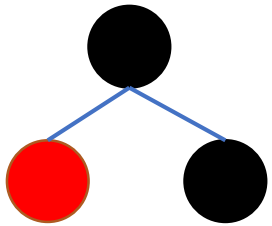
- 一般来讲，set中的元素是互不相同的，即使插入相同的元素，也只保留一个。
  - map的key值也是同理。
- 如果要实现在set中存储多个相同元素，就需要用到multiset

```
/* 声明 */
multiset<int> s;

/* 赋值 */
s.insert(9); s.insert(4);
s.insert(6); s.insert(4);
// set中有4个元素

/* 查找 */
cout << *s.find(4) << endl; // 输出4
cout << s.count(4) << endl; // 输出2
s.erase(s.find(4)); // 删除一个4
s.erase(5); // 删除所有的5
```

```
/* 声明 */
multimap<int,int> m;

/* 赋值 */
m.emplace(1,3); m.emplace(1,4);
// 插入两个键值对，其中键相同

/* 查找 */
cout << m.count(1); // 输出2
cout << m.find(1)->first << ' ' << m.find(1)->second;
// 输出1 3，实际上可能会输出任意一个键值对
m.erase(s.find(1)); // 删除一个1
m.erase(1); // 删除所有的1
```

基于红黑树的 map/set

- 红黑树 RB-Tree
  - C++ STL 中的 map/set 都是基于 RB-Tree 实现的，红黑树与之前学过的 AVL 树都是平衡树，但是红黑树不追求完全平衡，插入和删除的旋转次数较 AVL 树少，插入和删除的复杂度极优于 AVL 树。
  - 增删改查 的复杂度都是 log 级别
  - 并且，底层要求模板类<T>实现了比较方法



- 思考：
  - 利用红黑树来实现 map/set 好处是可以维护元素间的关系（有序性）
  - 但倘若不关心元素间的关系，map 的功能和数据结构上学的哈希表功能又一样，为什么不能要 O(1) 的增删改查性能?
  - —— 这便引入了基于哈希表的 unordered_map/set

基于红黑树/哈希表的 map/set

- 总结

	map/set	unordered_map/set
底层	红黑树	哈希开链法
元素间关系	元素有序，可从小到大遍历	元素无序，可自然遍历
需要重写方法	比较方法	判等方法、hash()方法
单次操作复杂度	log 级别	最好 O(1) 最坏 O(n)
总体复杂度	复杂度较稳定	大部分情况复杂度优 但“常数比较大”（hash()方法的常数耗时、哈希表的建立）

- map 的用法主要有三个 （了解）
  - 离散化数据
  - 判重与去重 (set也行)
  - 需要 logn 级别的 insert/delete 性能，同时维护元素有序

基于哈希表的 unordered_map/set

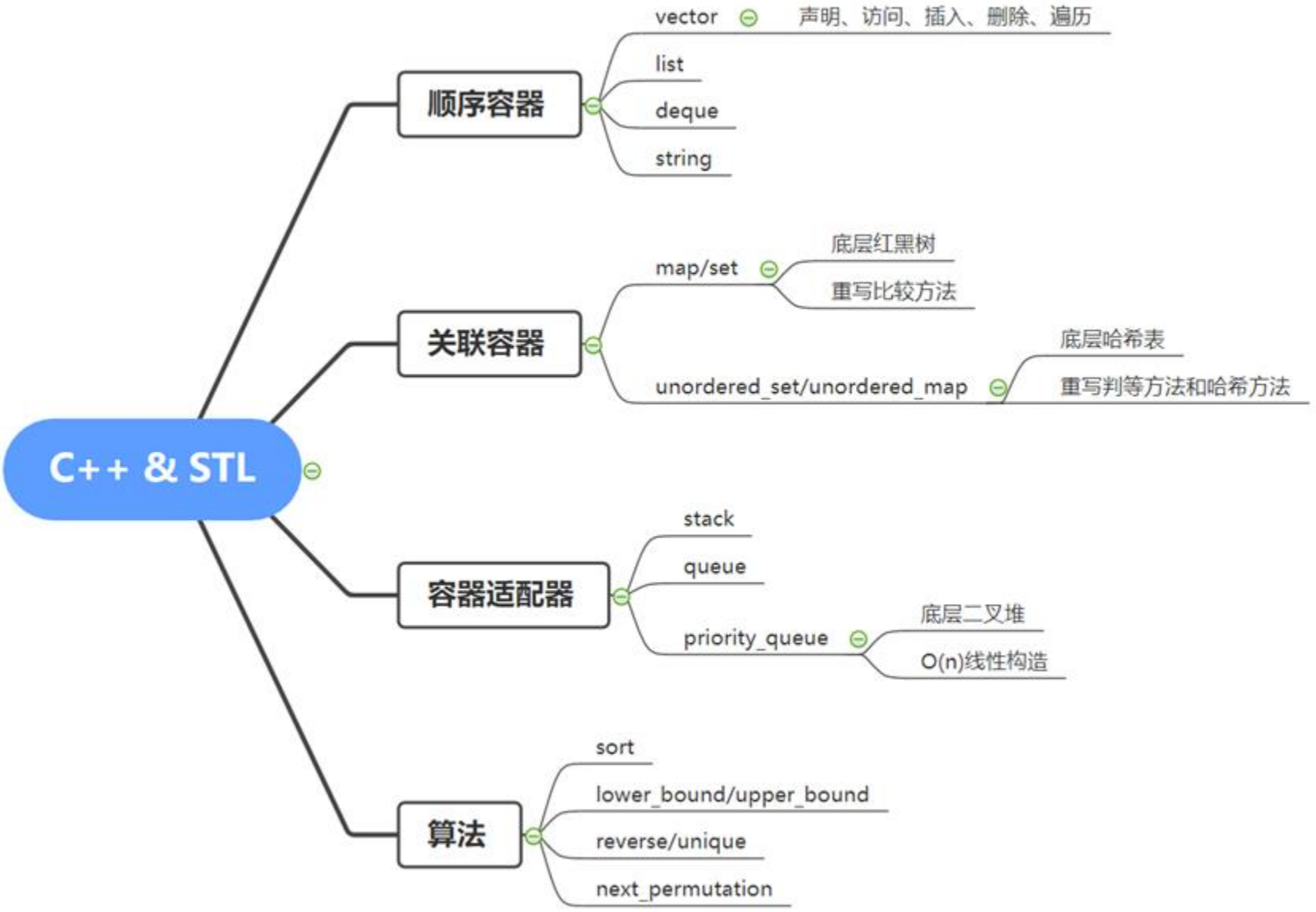
- 如果模板类<T> 是常用数据类型（int, char, bool 等），那么和 map / set 用法一样，只需要在声明时把 “map” 改为 “unordered_map”，就能把 O(logn) 的性能调为 O(1) ！
- 提示：**C++ 11 及其之后的版本才支持**
- 了解 重写 == 方法、hash() 方法
  - 正如基于红黑树的 map/set 要求元素类型<T>重写比较方法
  - 基于哈希表的 unordered_map / unordered_set 在底层进行复杂元素的判断时，要求实现 判等方法和 hash() 方法

基于哈希表的 unordered_map/set

- 了解 重写 == 方法、hash() 方法

```
struct Point {
    int x, y;
    Point() {}
    Point(int _x, int _y):x(_x), y(_y) {}
    bool operator == (const Point &t) const {
        return  x==t.x && y==t.y;
    }
};
struct PointHash {
    std::size_t operator () (const Point &p) const {
        return  p.x * 100 + p.y;    // 如果数据范围 x,y<100, hash 方法可以这样写
    }
};
int main()
{
    unordered_map<Point, bool, PointHash> mp;
    printf("%d\n", mp[{1,5}]);
    mp[{1,5}] = true;
    printf("%d\n", mp[{1,5}]);
    unordered_set<Point, PointHash> st;
    return 0;
}
```

总结


C++ & STL
顺序容器
vector
声明、访问、插入、删除、遍历
list
deque
string
关联容器
map/set
底层红黑树
重写比较方法
unordered_set/unordered_map
底层哈希表
重写判等方法和哈希方法
容器适配器
stack
queue
priority_queue
底层二叉堆
O(n)线性构造
算法
sort
lower_bound/upper_bound
reverse/unique
next_permutation

感谢收听

Thank You For Your Listening