



程序设计思维与实践

Thinking and Practice in Programming

C++ 与 STL I

内容负责：师浩晏

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
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	映射
	set	集合
	unordered_map	（了解 C++11）
	unordered_set	（了解 C++11）
容器适配器	stack	栈
	queue	队列
	priority_queue	优先队列（堆）



1

回顾 C++

Review c++

struct和class

- 从功能上说，struct和class几乎没什么区别
- 在不显式声明的情况下，struct成员默认为public，class默认为private
- 和c语言的struct不同，c++的struct可以定义成员函数，重载运算符等操作
- 在编写竞赛代码时，我们不需要考虑成员的访问权限，全部使用public就可以解决问题。因此通常使用struct更方便，若使用class还需要手动标注public成员

引用

- 一个变量的引用相当于给它起了个别名

```
int a;  
int &b = a;  
// 如果改变b, a也会变化, 反之亦然
```

- 函数传参时使用引用可以避免拷贝, 相当于直接操作实参

```
void swap(int &a, int &b){ // 可以实现交换  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
void parse(string &s){ // 避免拷贝影响性能  
    // ...  
}
```

运算符重载

- 可以让自定义的结构体像基本类型一样通过运算符计算，非常方便
- 如果不用运算符重载，那么需要定义一个函数才能完成计算：

```
struct Point{
    int x, y;
    Point(int X = 0, int Y = 0): x(X), y(Y){}

    Point add(const Point &p) const {
        return Point(x + p.x, y + p.y);
    }
};
```

- 把函数名换成 operator + 即可完成加法运算重载

```
struct Point{
    int x, y;
    Point(int X = 0, int Y = 0): x(X), y(Y){}

    Point operator + (const Point &p) const {
        return Point(x + p.x, y + p.y);
    }
};

int main(){
    Point a(1, 2), b(3, 4);
    Point c = a + b;
}
```

template

- 模板的作用是，写一份代码，可以适用于多种类型，减少代码冗余

```
int min_int(int a, int b){  
    if(a < b) return a;  
    else return b;  
}  
  
double min_double(double a, double b){  
    if(a < b) return a;  
    else return b;  
}
```

- 两个函数几乎完全相同，只是参数类型和返回值类型不同

```
template <class T>  
T min(T a, T b){  
    if(a < b) return a;  
    else return b;  
}
```

- 用模板参数T替代实际类型，使用时再实例化为实际的类型

template

- 实际进行程序设计竞赛过程中，我们不需要手动编写模板，只要做到熟练运用 c++ 自带的模板即可

- 模板函数的调用：

```
int a = 10, b = 15;  
int c = max(a, b);
```

- 在调用模板函数的时候，编译器会根据传入参数类型自动判断模板参数类型
- 模板类：

```
map<int, string> mp;
```

- 需要在类名后面写一个尖括号，里面是模板参数

<bits/stdc++.h>

- 包含了c++标准库的所有头文件
- 平台相关，不是所有平台都支持
- 我们的oj环境，和csp的比赛环境都支持这个头文件
- 缺点：
 - 不是所有编译器都支持，影响编写代码的跨平台性
 - 让编译时间变慢
- 比赛环境比较确定，使用它可以只写一行include，节约时间；不用记忆每个类或函数在哪个具体头文件内
- 同样，由于竞赛代码为一次性代码，不用担心命名空间污染，所以直接在开头加一句 using namespace std;非常方便

```
#include<bits/stdc++.h>
using namespace std;
```

pair

- pair是一个结构体，包含两个成员，分别是first和second。两个模板参数分别标注两个成员的类型

```
pair<string, int> p;
```

- 相当于

```
struct s{  
    string first;  
    int second;  
} p;
```

- pair重载了< == >等运算符，先比较第一个成员，若第一个成员相同再比较第二个

```
pair<int, string> p1(2, "aab"), p2(3, "aaa"), p3(2, "aaa");  
p1 < p2; // true 因为 2 < 3  
p1 < p3; // false 因为 2 == 2 并且 "aab" > "aaa"
```

- 可以使用make_pair()或大括号 (c++11) 得到一个pair

```
pair<int, int> a, b;  
a = make_pair(3, 5);  
b = {3, 5};
```

tuple (了解)

- 如果成员不止两个，可以使用pair嵌套，也可以使用tuple，更方便

```
tuple<int, double, string> p; // 三个成员的tuple  
tuple<int, int, int, int> q; // 四个成员的tuple  
// tuple 允许更多成员，不再举例
```

- 使用get<N>来访问从0数第N个元素
- N必须是常量

```
get<2>(p); // 访问p的第二个成员
```

- 同样重载了各种比较方法



2

算法

algorithm

<algorithm>

- <algorithm>中提供了许多常用算法
- sort
- 二分相关
- 去重
- 全排列

sort

- 时间复杂度 $O(n\log n)$ 实现对指定区间进行排序
- 基础用法：
 - 传入区间第一个元素的地址
 - 和区间最后一个元素的下一个位置的地址
 - 将区间进行升序排序



```
int a[9] = {1,9,2,8,3,7,4,6,5};  
sort(a, a + 9);  
// a: {1,2,3,4,5,6,7,8,9}
```


sort

- 排序方式：函数的第三个参数
- 默认排序方式是升序排序，如果想进行降序排序或者自定义排序，可以修改第三个参数
- 第三个参数接受一个函数，这个函数要求传入两个元素，如果第一个应该排在第二个后面，则返回true，否则返回false
- 以下操作完成对整数的降序排序

```
bool cmp(int a, int b){  
    return a > b;  
}  
int a[9] = {1,9,2,8,3,7,4,6,5};  
sort(a, a + 9, cmp);  
// a: {9,8,7,6,5,4,3,2,1}
```

- c++本身提供了一个模板，greater<T>，它重载的小括号运算符也可以实现以上功能，这样进行降序排序更加方便

```
sort(a, a + 9, greater<int>());
```

- sort的第三个参数默认为less<T>(), 相当于自动调用小于号进行比较，进行升序排序

sort

- 如果对结构体进行排序，可以传入比较函数，也可以重载结构体的小于号，这样sort会自动根据小于号进行比较

```
struct s{
    int a, b, c;
    bool operator < (const s &x) const {
        //先按照a比较
        if(a < x.a) return true;
        if(a > x.a) return false;

        //若a相同再按照b比较
        if(b > x.b) return true;
        if(b < x.b) return false;

        //若ab均相同按照c比较
        return c < x.c;
    }
};

s arr[10];
/*
    进行赋值
*/
sort(arr, arr + 10); // 自动调用小于运算符，按照指定规则排序
```

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;
```

```
});
```

二分相关

- lower_bound和upper_bound
- 传入一个区间（第一个位置和最后一个元素的下一个位置）和一个值
- 假定最后一个元素的下一个位置是无穷大
- lower_bound可以在指定区间内查找大于等于给定值的第一个位置
- upper_bound可以在指定区间内查找大于给定值的第一个位置

```
int a[10] = {1,2,3,4,5,5,5,6,7,8}
lower_bound(a, a + 10, 5) - a; // 4
upper_bound(a, a + 10, 5) - a; // 6
```

- 讲到二分时会详细展开

unique

- unique用于去重，对于连续相同元素，只保留一个
- 由于去重后元素数量会变少，unique函数会返回去重后新的区间结束位置

```
int a[10] = {1,1,2,3,3,5,3,3,4,1};  
unique(a, a + 10); //返回a + 7, a到a + 7区间内变成1,2,3,5,3,4,1
```

- 这个函数经常配合sort使用，因为sort会让区间内相同的元素放在一起

reverse, min_element, max_element

- reverse: 传入一个区间（左闭右开），对区间内元素进行反转
- 1, 2, 3, 4, 5 \rightarrow 5, 4, 3, 2, 1
- min_element/max_element: 返回区间内最小/最大值
- 这两个函数比较简单，即使记不住也可以手动写循环解决问题

next_permutation

- 生成指定区间的下一个排列，若不存在下一个排列则返回false

```
#include<bits/stdc++.h>
using namespace std;
const int n = 4;
int a[n];
int main(){
    for(int i = 0; i < n; i++){
        a[i] = i;
    }
    do{
        for(int i = 0; i < n; i++){
            cout << a[i] << ' ';
        }
        cout << '\n';
    }while(next_permutation(a, a + n));
    return 0;
}
```



3

顺序容器

Sequence container

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

```
vector<int> a; // 定义一个vector，类型int，长度为0
vector<int> b(10); // 定义一个vector，类型int，长度为10
vector<int> c(10, 3); // 定义一个vector，类型int，长度为10，并且初始值均为3
```

```
vector a(10);
a[2] = 5;
// a[10] = 3; 错误，越界，最大可以访问到a[9]
a.back() = 4; // 将a中最后一个数改成4
a.clear(); // a的长度变为0，清空
```

```
vector<int> a(10);
a.push_back(3); // a的末尾增加一个元素3，尺寸增大了1
a.pop_back(); // 把a末尾的元素删除。如果a本身长度为0会产生错误
```

```
vector<int> a = {1,2,3,4,5,6,7,8,9}; // 使用初始化列表构造
a.resize(11); // a = {1,2,3,4,5,6,7,8,9,0,0};
a.resize(13, 5); // a = {1,2,3,4,5,6,7,8,9,0,0,5,5};
a.resize(7); // a = {1,2,3,4,5,6,7}
```

```
int x = a.size(); // 返回一个整数，为a的长度
bool y = a.empty(); // 返回一个bool，若a长度为0则返回true，否则返回false
```

vector

- 迭代器
- 先回顾一下什么是迭代器
- 迭代器，其实类似一个指针，只不过指针指向的是一个物理地址，而迭代器，是一个虚拟的结构，指向的数据结构中的一个位置。可以通过迭代器访问它所指的内容，也可以根据迭代器访问数据结构中相邻的内容，这一点类似指针的++，--，[]操作
- begin和end分别返回指向vector第一个元素的迭代器和指向最后一个元素的下一个位置的迭代器，类型为vector<T>::iterator

```
sort(a.begin(), a.end());
sort(a.begin() + 1, a.end()); // 忽略第0个元素，对其他进行排序
```

- 可以使用下标遍历，也可以使用迭代器遍历

```
for(int i = 0; i < a.size(); i++){
    // a[i] 为访问的元素
}
for(vector<int>::iterator i = a.begin(); i != a.end; i++){
    // *i 即为我们访问的元素
}
for(int &i: a){ // int 可以替换为 auto
    // i 本身就是访问元素的引用
} // c++ 11的 range for写法
```

vector

- 插入删除（了解）
- vector可以在任意位置进行插入删除元素操作，时间复杂度为 $O(n)$
- 在某个迭代器前面插入一个元素

```
vector<int> a = {1,3,5,7,9}  
a.insert(a.begin() + 2, 0); // a = {1,3,0,5,7,9}
```

- 删除某个迭代器所指的元素

```
vector<int> a = {1,3,5,7,9}  
a.erase(a.begin() + 2); // a = {1,3,7,9}
```

- 删除一个区间（左闭右开）

```
vector<int> a = {1,2,3,4,5,6,7,8,9}  
a.erase(a.begin() + 1, a.begin() + 4); // a = {1,5,6,7,8,9}
```

- 一切会让vector长度增加的操作，都有可能让之前的迭代器失效

list(了解)

- 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。

```
list<int> l1;

l1.push_back(1); // 末尾增加一个元素
l1.push_front(1); // 最前面插入一个元素
l1.pop_front(); // 删除第一个元素
l1.pop_back(); // 删除最后一个元素
// 均为  $O(1)$  时间复杂度

list<int>iterator it = l1.begin();
it++; // 迭代器移动
l1.insert(it, 1); // 在it之前插入一个新元素
it = l1.erase(it); // 删除it所在位置的元素，返回被删除元素的下一个元素的迭代器。原先迭代器由于指向位置被删除而失效
// 均为  $O(1)$  时间复杂度

l1.clear(); // 删除所有元素，时间复杂度为  $O(n)$ 

l1.sort(); // 对l1的内容进行升序排列
l1.unique(); // 对l1的内容进行相邻去重
```


list (了解)

- 可以使用迭代器或 c++ 11 的 range for 进行遍历

```
for(auto i = l1.begin(); i != l1.end(); i ++){  
    cout << *i << ' ';  
}  
  
for(auto &i: l1){  
    cout << i << ' ';  
}
```

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 s1 = "123";
string s2;
cin >> s2;
cout << s1 << '\n'; // 可以使用 cin 和 cout 输入和输出string

cout << s1.size() << '\n'; // 长度为3

string s3 = "abc";

s1 += s3; // s1 = "123abc"
s1 += 'x' // s1 = "123abcx"
cout << s1 << '\n';

s2 = s1 + s3; // s2 = 123abcxabc
```

```
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
```



4

关联容器

Associative containers

set

下面先关注用法，再了解原理

- set可以维护一个集合，可以增加元素，删除元素，查找元素，同时保证元素唯一性
- 以上操作时间复杂度都是 $O(\log n)$

```
set<int> s; // 定义一个set，里面装整数。一开始是空集

s.insert(3); // s中插入3
s.insert(3); // s中又插入3，但没有实际效果，因为本来就有了

s.erase(3); // 删除s中的3。现在s又是一个空集了
s.erase(4); // 删除s中的4。但没有实际效果，因为s中本来就没有4
// insert 和 erase 的复杂度均为 $O(\log n)$ 
```

查找操作将返回一个迭代器，若找到则返回的迭代器指着这个元素，找不到则返回end()

```
set<int> s = {1,2,3,4,5,7,8};
auto it1 = s.find(4), it2 = s.find(6); // 这里的auto可以改成set<int>::iterator
if(it1 == s.end()) cout << "4 is not found\n";
if(it2 == s.end()) cout << "6 is not found\n";
```

set

下面先关注用法，再了解原理

- 可以使用count函数判断一个元素是否存在

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

- 遍历

```
set<int> s = {1,2,3,4,5,6,7,8,9};  
for(auto i = s.begin(); i != s.end(); i++){  
    cout << *i << ' ';  
}  
cout << '\n';  
  
for(const int &i: s){ // const int 可改为 auto  
    cout << i << ' ';  
}
```


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：

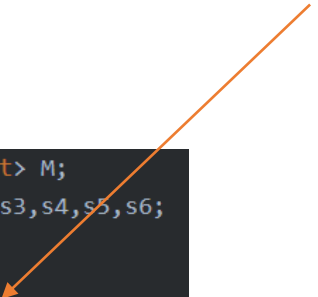
```
/* 清空 */
```

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

```
mp.clear();
```

```
/* 常量map声明，而不是声明一个空的map随后在main中赋值 */
```

```
const 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;
}
```

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
```

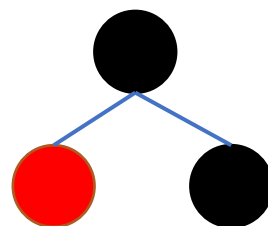
```
/* 声明 */
multimap<int,int> m;

/* 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 $O(\log n)$ 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;
}
```

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 $\log n$ 级别的 insert/delete 性能，同时维护元素有序



5

容器适配器

Container adaptors

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});
```

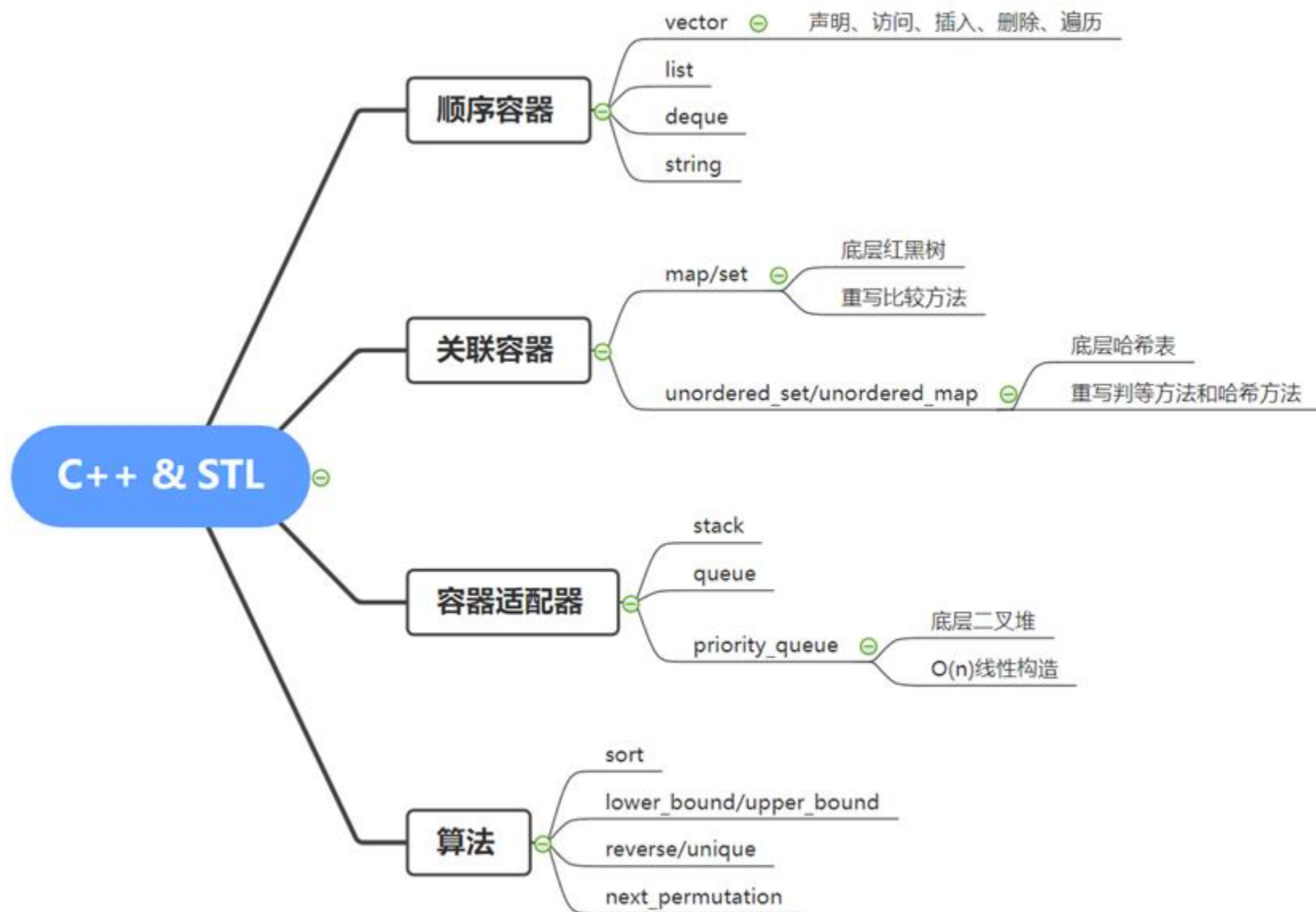
```
heap.push({1, 3, 4});
```

```
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<号可以代替其他所有比较
- $a > b$ 可以用 $b < a$ 替代
- $a \leq b$ 可以用 $!(b < a)$ 替代
- $a \geq b$ 可以用 $!(a < b)$ 替代
- $a == b$ 可以用 $!(b < a) \ \&\& \ !(a < b)$ 替代
- $a != b$ 可以用 $a < b \ || \ b < a$ 替代

总结





为天下储人才
为国家图富强

感谢收听

Thank You For Your Listening