
第10课 STL、IO流

林 评

内容概述

1. 算法概述
2. 常用算法
3. 排序算法
4. 二分搜索算法
5. 特定容器算法
6. 练习1
7. 关联容器 set map
8. 练习2
9. 无序容器 unordered_set
unordered_map
10. C语言文件操作
11. IO流结构
12. fstream
13. stringstream
14. 缓冲区
15. 练习3

STL: 算法 find find_if count count_if

```
//查找find,find_if,count,count_if
```

```
#include <iostream>
#include <vector>
#include <string>
#include <list>
#include <array>
#include <algorithm>
using namespace std;
```

```
struct cmp1 {
    cmp1(int th=0):thresh(th){}
    bool operator()(const int& t) {
        if (t > thresh) return true;
        return false;
    }
    int thresh;
};

struct cmp2 {
    cmp2(double th = 0) :thresh(th) {}
    bool operator()(const double& t) {
        if (t == thresh) return true;
        return false;
    }
    double thresh;
};
```

```
vector<int> v1 = { 2,1,2,4,7, };
//find(beg,end,val) beg:迭代器起始位置,end:迭代器结束位置,val:要找的值
//在迭代器范围内,找到该元素,则: 返回第一个该元素的迭代器位置
vector<int>::const_iterator res1 = find(v1.cbegin(), v1.cend(), 1);
if (res1 != v1.end())
    cout << "元素1:找到," << *res1 << endl;
//若没有找到该元素,则: 返回end位置的迭代器
vector<int>::iterator res2 = find(v1.begin() + 3, v1.end() - 1, 10);
if (res2 == v1.end() - 1) //vector迭代器是随机迭代器,可以+ -
    cout << "元素10:没有找到\n";
```

```
//find_if(beg,end,unaryPred) beg:迭代器起始位置,end:结束位置,unaryPred:条件
//在迭代器范围内,找符合条件unaryPred的元素,找到、未找到返回同find()
//unaryPred: 一元谓词,接受一个参数(就是迭代器所指向的元素),返回可作为条件的类型
auto res3 = find_if(v1.cbegin(), v1.cend(), cmp1(4)); //找>4的元素
if (res3 != v1.cend())
    cout << ">4的元素找到," << *res3 << endl;
auto res4 = find_if(v1.begin(), v1.end(), cmp1(10)); //找>10的元素
if (res4 == v1.end())
    cout << ">10的元素没有找到\n";
```

```
//count(beg,end,val)
//在迭代器范围内,找出元素值==val 的元素个数,返回值等于val的元素个数
list<double> l1 = { 1.0,2.0,1.0,3.0 };
cout << count(l1.begin(), l1.end(), 1.0) << endl; // 2
cout << count(l1.begin(), l1.end(), 4.0) << endl; // 0

//count_if(beg,end,unaryPred)
//在迭代器范围内,统计符合条件unaryPred的元素个数
cout << count_if(l1.cbegin(), l1.cend(), cmp2(1.0)) << endl; //==1.0的个数 2
```

```
元素1:找到,1
元素10:没有找到
>4的元素找到,7
2
0
2
请按任意键继续.
```

STL: 算法 search

//查找子序列 search

```
#include <iostream>
#include <vector>
#include <string>
#include <list>
#include <array>
#include <algorithm>
using namespace std;

bool cmp3(const int& a, const int& b) {
    //只要都是奇数 或 偶数 就认为相等
    if (a % 2 == b % 2) return true;
    return false;
}
```

```
找到子序列2,3,5: 2
没有找到子序列2,4
3,4,1,5子序列找到: 1
4,1,3,2子序列找到: 8
1,1,2,1子序列没找到
1,1,2,1的反向子序列找到: 1
请按任意键继续. . .
```

```
//search(beg1,end1,beg2,end2) beg1,end1和beg2,end2表示2对迭代器范围
//返回 beg2到end2(子序列) 在beg1到end1中间第1次出现的位置,没有找到返回end1
int arr1[] = { 1,2,3,5,8,2,3,5 };
int arr2[] = { 2,3,5 }, arr3[] = { 1,2,4 };
auto res5 = search(begin(arr1), end(arr1), begin(arr2), end(arr2));
if (res5 != end(arr1))
    cout << "找到子序列2,3,5: " << *res5 << endl; /*res = 2
auto res6 = search(cbegin(arr1), cend(arr1), &arr3[1], &arr3[3]);
if (res6 == cend(arr1))
    cout << "没有找到子序列2,4\n";
```

```
//search(beg1,end1,beg2,end2,binaryPred)
//binaryPred: 二元谓词,接受两个参数(两个元素的比较),返回可作为条件的类型
//查找beg2到end2子序列 在beg1到end1中间 出现的位置,判断条件用 binaryPred
array<int, 5> a1 = { 1,8,3,3,4 }; //奇 偶 奇 奇 偶
array<int, 4> a2 = { 3,4,1,5 }, a3 = { 4,1,3,2 }, a4 = { 1,1,2,1 };
auto res7 = search(a1.begin(), a1.end(), a2.begin(), a2.end(), cmp3);
if (res7 != a1.end())
    cout << "3,4,1,5子序列找到: " << *res7 << endl;
auto res8 = search(a1.begin(), a1.end(), a3.begin(), a3.end(), cmp3);
if (res8 != a1.end())
    cout << "4,1,3,2子序列找到: " << *res8 << endl;
auto res9 = search(a1.begin(), a1.end(), a4.begin(), a4.end(), cmp3);
if (res9 == a1.end())
    cout << "1,1,2,1子序列没找到\n";
auto res10 = search(a1.cbegin(), a1.cend(), a4.cbegin(), a4.crend(), cmp3);
if (res10 != a1.end())
    cout << "1,1,2,1的反向子序列找到: " << *res10 << endl;
```

STL: 算法 for_each fill fill_n

```
//for_each, fill,fill_n (插入迭代器)
#include <iostream>
#include <vector>
#include <string>
#include <forward_list>
#include <deque>
#include <array>
#include <algorithm>
using namespace std;
class Stu {
public:
    friend ostream& operator<<(ostream&, const Stu&);
    Stu(const string& _name,int _age)
        :name(_name),age(_age){}
private:
    string name;
}; int age;
ostream& operator<<(ostream &out,const Stu& s) {
    out << s.name << ":" << s.age;
} return out;
int main() {
    //for_each(beg,end,unaryOp) unaryOp:一元操作
    //对迭代器beg到end之间的每个元素执行 unaryOp操作
    forward_list<int> f1 = { 1,2,3 };
    for_each(f1.cbegin(), f1.cend(), //打印输出
        [](const int& item) {cout << item << " -> "; });
    cout <<"NULL"<< endl;
    //如果迭代器允许通过解引用改变元素值,unaryOp也可修改元素值
    for_each(f1.begin(), f1.end(), [](int &item) {if (item > 1)item+=10; });
    for (const auto& item : f1)
        cout << item << " -> ";
    cout << "NULL" << endl;
```

```
1 -> 2 -> 3 -> NULL
1 -> 12 -> 13 -> NULL
张三:21 李四:22 王五:20
NULL:0 NULL:0 NULL:0
4
0 0 0 4 5 6 7
0 0 0 4 5 6 7 11 11 11
王五:20 王五:20 NULL:0 NULL:0 NULL:0
0 0 0 4 5 6 7 11 11 11 22 22 22
-1 -1 -1 0 0 0 4 5 6 7 11 11 11 22 22 22
请按任意键继续. . .
```

```
//fill(beg,end,val)
//将迭代器范围beg到end中的元素值 设置为val
deque<Stu> d1 = { {"张三",21},{ "李四",22},{ "王五",20} };
for (const auto& item : d1) cout << item << " "; cout << endl;
fill(d1.begin(), d1.end(), Stu("NULL", 0));
for (const auto& item : d1) cout << item << " "; cout << endl;
```

```
//fill_n(beg,cnt,val) cnt个数
//从迭代器beg位置开始,设置cnt个元素的值为val,返回最后一个设置元素后面位置
vector<int> v1 = { 1,2,3,4,5,6,7 }; //的迭代器
auto res1 = fill_n(v1.begin(), 3, 0); //从开始位置,设置3个0
cout << *res1 << endl; //4
for (const auto& item : v1) cout << item << " "; cout <<endl;
//fill_n(v1.begin(), 10, 1); 错误,v1没有10个元素,程序员自己保证
```

```
//back_inserter: 插入迭代器(迭代器适配器) 使用push_back的迭代器
back_inserter_iterator<vector<int>> it_back = back_inserter(v1);
fill_n(it_back, 3, 11); //v1后面增加3个11(调用push_back)
for (const auto& item : v1) cout << item << " "; cout << endl;
```

```
//front_inserter: 插入迭代器(迭代器适配器) 使用push_front的迭代器
//fill_n(front_inserter(v1), 2, 33); 错,vector没有push_front
fill_n(front_inserter(d1), 2, Stu("王五", 20)); //调用push_front
for (const auto& item : d1) cout << item << " "; cout << endl;
```

```
//inserter: 插入迭代器(迭代器适配器) 在给出的迭代器位置前插入
inserter_iterator<vector<int>> it_insert = inserter(v1, v1.end());
fill_n(it_insert, 3, 22);
for (const auto& item : v1) cout << item << " "; cout << endl;
fill_n(inserter(v1, v1.begin()), 3, -1);
for (const auto& item : v1) cout << item << " "; cout << endl;
```


STL: 算法 copy copy_if copy_n transform

```
//copy,copy_if,copy_n,transform
#include <iostream>
#include <vector>
#include <string>
#include <deque>
#include <array>
#include <list>
#include <forward_list>
#include <algorithm>
using namespace std;
class Stu {
public:
    friend ostream& operator<<(ostream&, const Stu&);
    Stu(const string& _name, int _age)
        :name(_name), age(_age) {}
private:
    string name;
    int age;
};
ostream& operator<<(ostream &out, const Stu& s) {
    out << s.name << ":" << s.age;
    return out;
}
int main() {
    //copy(begin,end,dest) dest:存放处理后元素的目标位置
    //将序列中的元素拷贝到dest指定的目标序列中
    //dest下要能放得下那么多元素
    int arr[] = { 11,22,33 };
    vector<int> v0(5);
    copy(begin(arr), end(arr), v0.begin()); //arr拷贝到v0
    for (const auto &item : v0) cout << item << " "; cout << endl;
```

```
//copy_if(begin,end,dest,unaryPred) unaryPred:一个参数的条件
//只有符合条件的才拷贝(unaryPred返回true表示符合条件)
copy_if(begin(arr), end(arr), v0.begin(), //拷贝奇数
        [](const int& t) {if (t % 2 == 1) return 1; return 0; });
for (const auto &item : v0) cout << item << " "; cout << endl;
//copy_n(begin,end,n,dest)
//复制前n个元素,begin开始的序列中要有n的元素,dest中能放下n个元素
vector<int> v00; //用back_inserter
copy_n(begin(arr), sizeof(arr) / sizeof(int), back_inserter(v00));
for (const auto &item : v00) cout << item << " "; cout << endl;
cout << "-----\n";
//transform(begin,end,dest,unaryOp)
//将序列中的元素调用unaryOp操作,结果存入dest
list<int> l11 = { 1,2,3,4,5,6 };
vector<int> v11(10, 0); //将list中的元素+10后存入vector
transform(l11.cbegin(), l11.cend(), v11.begin(), //每个元素+10
/*要保证dest里面的位置足够*/ [](const int& t) {return t + 10; });
for (const auto &item : v11) cout << item << " "; cout << endl;
vector<int> v12; //利用 back_inserter 插入迭代器实现
transform(l11.cbegin(), l11.cend(), back_inserter(v12),
        [](const int& t) {return t + 10; });
for (const auto &item : v12) cout << item << " "; cout << endl;
//transform(begin,end,begin2,dest,binaryOp) binaryOp:两个参数的操作
//将两个序列中的元素作为2个参数传递给binaryOp,操作结果存入dest
array<int, 3> a13 = { 21,22,20 };
forward_list<string> f13 = { "张三", "李四", "王五" };
deque<Stu> d13;
transform(a13.cbegin(), a13.cend(), f13.cbegin(), back_inserter(d13),
        [](const int& age, const string& name) {return Stu{name,age}; });
for (const auto &item : d13) cout << item << " "; cout << endl;
```

STL: 算法概述

顺序容器中定义的操作: 添加、删除、访问首尾元素等等。

对于查找、替换、删除特定的元素、重排元素等大多没有通过成员函数实现, 而是定义了一组“泛型算法”
标准库定义了超过100种算法, 大多数算法在<algorithm>中, 数值算法在<numeric>中。

迭代器令算法不依赖于容器, 但算法依赖于元素类型的操作。(如: find要求元素有==运算)

算法不会执行容器的操作, 只是运行于迭代器之上。算法不会改变底层容器的大小, 有可能改变元素的值, 有可能在容器内移动元素, 但不会直接添加或删除元素。

beg,end: 迭代器范围 **beg2,end2**: 第2个输入序列迭代器(end2有可能没有) **dest**: 目的序列迭代器
unaryPred,binaryPred: 一元、二元条件(返回真假) **comp**: 二元比较 **unaryOp,binaryOp**: 一元、二元操作

迭代器适配器: 插入迭代器 (back_inserter, front_inserter, inserter) #include <iterator>

输入迭代器: 只读, 不写; 单遍扫描; 只能递增

输出迭代器: 只写, 不读; 单遍扫描; 只能递增

前向迭代器: 可读写; 多遍扫描; 只能递增

双向迭代器: 可读写; 多遍扫描; 可递增递减

随机访问迭代器: 可读写; 多遍扫描; 支持全部迭代器运算

STL: 算法 排序

排序算法：需要**随机访问迭代器**。所以：list, forward_list不能用该算法。通过成员函数实现。
快排、堆排序属于不稳定排序，归并排序是稳定排序。

```
//sort(beg,end) 排序,要求随机访问迭代器,元素有<运算,从小到大  
//sort(beg,end,comp) 用comp来确定如何排序  
vector<int> v1{ 2,1,3,4,2,8,9,0 };  
sort(v1.begin(), v1.end());  
for (const auto& item : v1) cout << item << " "; cout << endl;  
sort(v1.begin(), v1.end(), greater<int>());  
for (const auto& item : v1) cout << item << " "; cout << endl;  
sort(v1.begin(), v1.end(), //奇数在前,偶数在后  
    [](const int& t1, const int& t2){if(t2%2 < t1%2)return 1;return 0;});  
for (const auto& item : v1) cout << item << " "; cout << endl;  
cout << "-----\n";
```

```
//stable_sort(beg,end) 稳定排序,随机访问迭代器, <运算  
//stable_sort(beg,end,comp)  
stable_sort(v1.begin(), v1.end());  
for (const auto& item : v1) cout<<item<<" ";cout<<endl;  
cout << "-----\n";
```

```
0 1 2 2 3 4 8 9  
9 8 4 3 2 2 1 0  
9 3 1 8 4 2 2 0  
-----  
0 1 2 2 3 4 8 9  
1 1 2 3 9 8  
9 8 3 2 1 1  
-----  
it1: 5  
1 2 3 4 5 8 9  
请按任意键继续.
```

```
//partial_sort(beg,mid,end) 排序mid-beg个元素,相当于找出前几个最小元素  
//partial_sort(beg,mid,end,comp)  
deque<int> d1 = { 9,8,1,3,2,1 };  
partial_sort(d1.begin(), d1.begin() + 4, d1.end()); // 1 1 2 3...  
for (const auto& item : d1) cout << item << " "; cout << endl;  
partial_sort(d1.begin(),d1.begin()+4,d1.end(),greater<int>());//9 8 3...  
for (const auto& item : d1) cout << item << " "; cout << endl;  
cout << "-----\n";
```

```
//nth_element(beg,nth,end) nth是迭代器位置(指向一个元素),  
//排序完后,nth指向还是那个元素,迭代器之前的都小于等于它,迭代器之后的都大于等于它  
//nth_element(beg,nth,end,comp)  
array<int, 7> a1 = { 1,3,8,2,5,4,9 };  
auto it1 = a1.begin() + 4; //指向元素 5  
nth_element(a1.begin(), it1, a1.end());  
cout << "it1: " << *it1 << endl;  
for (const auto& item : a1) cout << item << " "; cout << endl;
```


STL: 算法 二分搜索

二分搜索要求已序序列，要求至少是前向迭代器(如forward_list)，但是提供随机访问迭代器会提升很多的性能。

无论前向还是随机访问迭代器，都是logN次的比较操作，但是前向迭代器就要消耗线性时间来操作迭代器移动到要比较的元素位置。

```
//binary_search(beg,end,val) 在已序序列上搜索是否有元素值==val
//有返回true,没有返回false 【必须有 < (==) 运算】
//binary_search(beg,end,val,comp)
vector<int> v1{ 20,10,30,40,20,80,90,0 };
sort(v1.begin(), v1.end());
bool flag = binary_search(v1.begin(), v1.end(), 31);
cout << flag << endl;
flag = binary_search(v1.begin(), v1.end(), 31, //误差+-1认为相等
    [](const int&t1, const int&t2){if(t1+1==t2||t1-1==t2)return 1;return 0;});
cout << flag << endl; //31被找到了。
cout << "-----\n";
```

```
//lower_bound(beg,end,val) 在已序序列上搜索元素==val,没有搜到返回end
//找到,返回一个迭代器:指向第一个大于等于 val 的元素 【<运算】
//lower_bound(beg,end,val,comp)
v1 = { 1,2,3,4,4,4,5,6 };
auto it1 = lower_bound(v1.begin(), v1.end(), 3); //找3,返回3的位置
cout << "找3: " << *it1 << endl;
it1 = lower_bound(v1.begin(), v1.end(), 4); //找4,返回第1个4的位置
for_each(it1, v1.end(), [](const int& t) {cout << t << " "; });
cout << endl <<"-----\n";
```

```
//upper_bound(beg,end,val) 在已序序列上搜索元素==val,没有搜到返回end
//找到,返回一个迭代器:指向第一个大于 val 的元素
//upper_bound(beg,end,val,comp)
auto it2 = upper_bound(v1.cbegin(), v1.cend(), 3); //找3,返回第1个4的位置
for_each(it2, v1.cend(), [](const int& t) {cout << t << " "; });
cout << endl;
it2 = upper_bound(v1.cbegin(), v1.cend(), 4); //找4,返回第5的位置
for_each(it2, v1.cend(), [](const int& t) {cout << t << " "; });
cout << endl << "-----\n";
```

```
//equal_range(beg,end,val) 在已序序列中找元素==val,返回一对pair,
//first成员是lower_bound返回的迭代器,second成员是upper_bound返回的迭代器
//equal_range(beg,end,val,comp)
for (const auto& item : v1) cout << item << " "; cout << endl;
pair<vector<int>::iterator, vector<int>::iterator> pr1 =
    equal_range(v1.begin(), v1.end(), 4);
cout << "*first: " << *pr1.first << endl;
cout << "*second: " << *pr1.second << endl;
for_each(pr1.first, pr1.second, [](const int &t) {cout << t << " "; });
cout << endl;
```

STL: 算法 remove unique

```
//remove(beg,end,val) 删除值是val的元素 ==运算
//算法不会改变容器的大小,采用的办法是:用保留的元素覆盖要删除的元素
//返回一个迭代器,指向最后一个删除元素的后面的位置
array<int, 5> a11 = { 1,2,3,1,4 };
auto it1 = remove(a11.begin(), a11.end(), 1);
cout << *it1 << endl;
for (const auto &item : a11) cout << item << " "; cout << endl;
```

```
//remove_if(beg,end,unaryPred) 根据unaryPred条件决定是否删除
forward_list<int> f11 = { 1,3,2,4,1 };
auto it2 = remove_if(f11.begin(), f11.end(), //偶数删除
    [](const int& t) {if (t % 2 == 0) return true; return false; });
cout << *it2 << endl;
for (const auto &item : f11) cout << item << " "; cout << endl;
cout << "-----\n";
```

```
//unique(beg,end) 对相邻元素,假如重复,会通过“覆盖”来删除(类似remove)
//返回一个迭代器,指向不重复元素的尾后位置 ==运算
//unique(beg,end,binaryPred)
array<int, 7> a1 = { 1,2,3,1,4,4,2 };
//不排序去重复:只会去除相邻的
for (const auto &item : a1) cout << item << " "; cout << endl;
auto it3 = unique(a1.begin(), a1.end());
for_each(a1.begin(), it3, [](const int& t) {cout << t << " "; });
cout << endl << "-----\n";
```

```
//排序以后,再unique:
a1 = { 1,2,3,1,4,4,2 };
sort(a1.begin(), a1.end());
for (const auto &item : a1) cout << item << " "; cout << endl;
auto it4 = unique(a1.begin(), a1.end());
for_each(a1.begin(), it4, [](const int& t) {cout << t << " "; });
cout << endl << "-----\n";
```

```
//reverse(beg,end) 翻转序列中的元素,需要双向迭代器
a1 = { 1,2,3,4,5,6,7 };
for (const auto &item : a1) cout << item << " "; cout << endl;
reverse(a1.begin(), a1.end());
for (const auto &item : a1) cout << item << " "; cout << endl;
cout << endl << "-----\n";
```

```
1
2 3 4 1 4
4
1 3 1 4 1
-----
1 2 3 1 4 4 2
1 2 3 1 4 2
-----
1 1 2 2 3 4 4
1 2 3 4
-----
1 2 3 4 5 6 7
7 6 5 4 3 2 1
-----
3 6 1 5 7 2 4
1 3 5 6 7 4 2
请按任意键继续
```

```
//random_shuffle(beg,end) 打乱序列 需要随机访问迭代器
random_shuffle(a1.begin(), a1.end());
for (const auto &item : a1) cout << item << " "; cout << endl;
random_shuffle(a1.begin(), a1.end());
for (const auto &item : a1) cout << item << " "; cout << endl;
```

STL: 特定容器算法

```
//list,forward_list特定算法(成员函数)
//1. reverse() 反转序列
forward_list<int> f1 = { 1,2,3,4,5 };
//reverse(f1.begin(), f1.end()); 不行,需要双向迭代器,list可以但效率不高
for (const auto& item : f1) { cout << item << " "; } cout << endl;
f1.reverse(); //反转
for (const auto& item : f1) { cout << item << " "; } cout << endl;
cout << "-----\n";

//2.remove(val),remove_if(pred) 删除 元素==val, pred条件为真
//remove(l1.begin(), l1.end(), 2);
f1.remove(2); //真正删除了元素2
for (const auto& item : f1) { cout << item << " "; } cout << endl;
f1 = { 1,2,3,4,5 };
remove(f1.begin(), f1.end(), 2); //虽然可用,但不是真正删除,效率低
for (const auto& item : f1) { cout << item << " "; } cout << endl;
cout << "-----\n";

//3.unique(),unique(pred) 去重复, == , pred条件为真 【真正去重,要先排序】
f1 = { 1,2,2,3,3,3,4,5,5 };
f1.unique(); //重复的删掉了
for (const auto& item : f1) { cout << item << " "; } cout << endl;
cout << "-----\n";

//4.sort(),sort(comp)
//sort(l1.begin(), l1.end()); 不行,sort算法要随机访问迭代器
f1 = { 3,2,1,2,3,4,9 };
f1.sort();
for (const auto& item : f1) { cout << item << " "; } cout << endl;
f1.sort([](const int& t1, const int& t2) {if (t1 > t2) return 1; return 0; });
for (const auto& item : f1) { cout << item << " "; } cout << endl; //从大到小
cout << "-----\n";
```

```
//5.merge(lst2), merge(lst2,comp) 都是已序的, <运算, comp返回为真
//合并2个已序序列
list<int> l1 = { 1,3,5,7,9 }, l2 = { 2,4,6,8,10 }, l3; //用算法merge: l2保留
merge(l1.begin(), l1.end(), l2.begin(), l2.end(), back_inserter(l3));
for (const auto& item : l3) { cout << item << " "; } cout << endl;

l1 = { 1,3,5,7,9 }; l2 = { 2,4,6,8,10 };
l1.merge(l2); //l2 内容被清空
for (const auto& item : l1) { cout << item << " "; } cout << endl;
for (const auto& item : l2) { cout << item << " "; } cout << endl;
cout << "-----\n";

//6.splice(p,lst2) splice_after(p,lst2)
//将lst2插入到p的位置 (对于forward_list就是p后的位置)
l1 = { 1,3,5,7,9 }; l2 = { 2,4,6,8,10 };
l1.splice(++l1.begin(), l2); //将l2插到3的位置,l2内容清空
for (const auto& item : l1) { cout << item << " "; } cout<<endl;
for (const auto& item : l2) { cout << item << " "; } cout<<endl;

forward_list<int> f0 = { 1,3,9,4,5 }, f2 = {13,11,12};
auto it1 = next(f0.before_begin(), 3); //it1在9的位置
f0.splice_after(it1, f2);
for (const auto& item : f0) { cout << item << " "; } cout<<endl;

//splice(p,lst2,p2) 将lst2中p2开始的元素移动到 当前列表的p位置
//splice(p,lst2,b,e) b,e代表lst2中的一个系列范围
//splice_after(p,lst2,p2)
//splice_after(p,lst2,b,e)
```

```
1 2 3 4 5
5 4 3 2 1
-----
5 4 3 1
1 3 4 5 5
-----
1 2 3 4 5
-----
1 2 2 3 3 4 9
9 4 3 3 2 2 1
-----
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
-----
1 2 4 6 8 10 3 5 7 9
-----
1 3 9 13 11 12 4 5
-----
请按任意键继续. . .
```

STL: 练习1

//练习1: 删除vector中特定的元素

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
//remove的自我实现
template <typename ForwardIterator, typename T>
ForwardIterator my_remove(ForwardIterator first,
                          ForwardIterator last, const T& val) {
    ForwardIterator result = first;
    while (first != last) {
        if (!(*first == val)) {
            *result = move(*first);
            ++result;
        }
        ++first;
    }
    return result;
}
template<typename T>
void remove_all_same_value(std::vector<T>& vec, const T& value) {
}

int main() {
    vector<int> v1 = { 1,2,3,2,2,1,4 };
    for (const auto& item : v1) cout << item << " "; cout << endl;
    remove_all_same_value(v1, 2);
    for (const auto& item : v1) cout << item << " "; cout << endl;
    return 0;
}
```

//利用std::remove实现

```
vec.erase(std::remove(vec.begin(), vec.end(), value), vec.end());
```

```
//实现 my_for_each,传入参数:容器,对容器执行操作
//for_each(beg,end,unaryOp) ==> my_for_each(容器的引用,unaryOp)
//自己写一个my_for_each
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

//找出下面代码的问题

```
template<typename C, typename F>
void my_for_each_1(C &c, F f) {
    for (auto it = c.begin(); it < c.end(); it++)
        f(*it);
}
```

```
int main() {
    vector<int> vec = { 1,2,3 };
    for_each(vec.begin(), vec.end(), [](int& v) {v += 1; });
    for_each(vec.begin(), vec.end(), [](int& v) {cout << v << " "; });
    cout << endl;

    vec = { 1,2,3 };
    my_for_each_1(vec, [](int& v) {v += 1; });
    my_for_each_1(vec, [](int& v) {cout << v << " "; });
    cout << endl;

    return 0;
}
```

//问题1: ++it (it++返回临时对象,效率低)

//问题2: it != c.end() (it < c.end() 有些容器迭代器不支持)

//问题3: for (auto it=c.begin(),it_end=c.end(); it!=it_end;it++) 效率会高一些

//问题4: 假如传入的是内置数组,所以: c.begin和c.end改写为std::begin(c)和std::end(c)

//问题5: 最简单写法 for_each(std::begin(c), std::end(c), f);

```
1 2 3 2 2 1 4
1 3 1 4
请按任意键继续
```


STL: 关联容器set/multiset(1)

```
//关联容器的元素是按关键字来保存和访问的。
//set(关键字不能重复), multiset(关键字可以重复) 头文件 #include <set> 红黑树
//template<class K,class Cmp=less<K>,class _Alloc=allocator<K>> class set...
//template<class K,class Cmp=less<K>,class _Alloc=allocator<K>> class multiset..
//1.元素自动排序
//2.插入和删除时间复杂度: O(logN) [需求: 频繁插入、删除、查找, 并且要求有序]
//3.元素必须支持“严格弱序”
// (1) (a < b)为真 ==> (b < a)为假, (a < a)肯定为假
// (2) (a < b) 并且 (b < c) ==> (a < c)
// (3) (a < b)为假 并且 (b < a)为假 ==> (a == b)
// (4) (a == b) 并且 (b == c) ==> (a == c)
//4.在set中的元素, 不能改变其值
//5.以Node(结点)的方式存放,
//内存消耗较大(左子树指针,右子树指针,父结点指针,前驱指针,后驱指针)

//pair的结构:
//template<typename T1, typename T2> struct pair {
// T1 first;
// T2 second;
// //... };
pair<int, double> p1(1, 2.0);
pair<int, char> p2 = { 10, 'c' };
pair<int, double> p3 = make_pair(1, 3.0);

//1.初始化
set<int,less<int>> g1; //默认就是 less<int>
multiset<int,greater<int>> g2;
set<double> g3{ 1.0, 3.0, 2.0, 2.0 }; //2.0只会存1次
set<double> g4{ 1.0, 3.0 }; //初始值列表
set<double> g5(g3);
set<double> g6 = g3; //拷贝构造
```

```
vector<int> vec = { 1,2,3,2,4,4,6 };
set<int> g7(vec.cbegin(), vec.cend()); //1 2 3 4 6
multiset<int> g8(vec.crbegin(), vec.crend()); //1 2 2 3 4 4 6
for (auto& item : g7) cout << item << " "; cout << endl;
for (auto& item : g8) cout << item << " "; cout << endl;
cout << "-----\n";
```

```
1 2 3 4 6
1 2 2 3 4 4 6
```

```
//2.赋值 (没有assign), swap
g5 = g6;
g7 = { 1,2,3,4,6 }; //初始值列表赋值
initializer_list<int> il = { 1,2,3,2,4,4,6 };
g8 = il;
g5.swap(g6); std::swap(g5, g6); //交换高效O(1)
```

```
//3.容量相关
g7.size(); //元素个数
g7.empty(); //元素个数==0则返回true
g7.max_size(); //能存储的最大元素个数
//以结点存放:没有capacity(),reserve(100),resize(25)
```

```
//4.元素访问: 不支持下标访问,at访问, 没有front,back这样的操作
//可通过迭代器遍历, 或者find查找元素
```

```
//5.迭代器相关
set<int>::iterator it1 = g7.begin(); //*it1不能赋值
set<int>::const_iterator it2 = g7.cbegin();
set<int>::reverse_iterator it3 = g7.rbegin();//*it3不能赋值
set<int>::const_reverse_iterator it4 = g7.crbegin();
//end()也是一样 实际上begin()返回的迭代器也是const的
```


STL: 关联容器set/multiset (2)

```
//6.特有算法: count,find,lower_bound,upper_bound,equal_range
//count: 标准算法的count是遍历的O(N)的,而set的是O(logN)
cout<<g7.count(1)<<" "<<g7.count(10)<<endl;//set只返回1或0
cout<<g8.count(2)<<endl; //multiset 返回>=0(有可能有多个)
//find: 标准算法的find是遍历
auto it5 = g7.find(1); //返回的是迭代器
if (it5 != g7.end()) //需要和end()比较才能知道是否有该元素
    cout << "g7: 1找到了\n";
else
    cout << "g7: 1没有找到\n";
//lower_bound,upper_bound,equal_range,标准算法最好是随机访问迭代器
auto it6 = g7.lower_bound(5); //返回第1个>=的元素位置
if (it6 != g7.end()) //g7: 1 2 3 4 6
    cout << "lower_bound 5: " << *it6 << endl; //6的位置
it6 = g7.upper_bound(5); //返回第1个>元素的位置
if (it6 != g7.end())
    cout << "upper_bound 5: " << *it6 << endl;//6的位置
//在multiset上用equal_range
pair<set<int>::iterator,set<int>::iterator> range_1=g8.equal_range(2);
auto range_2 = g8.equal_range(2); //用auto更加方便
//返回上面2个位置组成的pair
//return make_pair(g8.lower_bound(2),g8.upper_bound(2));
//实际上: 假如返回pair的first和second一致,说明没有找到该元素
//g8: 1 2 2 3 4 4 6
if (range_1.first!=g8.end()) cout<<"first: "<<*range_1.first<<" -- ";
else cout << "first: end() -- ";
if (range_1.second!=g8.end()) cout<<"second: "<<*range_1.second<<endl;
else cout << "second: end()\n";
cout << "-----\n";
```

```
//7.插入元素,没有push_back,emplace_back,push_front,emplace_front这样的操作
//只有insert 和 emplace 操作,可插入单个元素,也可插入迭代器一个范围的元素
g7.insert(10); //插入单个元素(返回值见后面)
g7.emplace(11); //传入的是元素类型的构造参数(返回值同上)
g7.insert(vec.begin(), vec.end()); //插入一个迭代器范围,返回void
g7.insert({ 1,3,9 }); //插入一个初始值列表,返回void
//返回值: 对于 set: 返回一个pair,first:插入元素的位置,second:true还是false
//对应插入set中已经存在的元素,返回pair的 second为false,插入操作不做任何事情
for (auto& item : g7) cout << item << " "; cout << endl;
auto r_pair = g7.insert(9); //9已经存在
cout << "插入元素:"<<r_pair.first <<" 是否成功:"<<r_pair.second << endl;
r_pair = g7.insert(8); //8不存在,会插入
cout << "插入元素:" << *r_pair.first <<" 是否成功:"<<r_pair.second<<endl;
for (auto& item : g7) cout << item << " "; cout << endl;
//返回值: 对于multiset,不存在插入失败的概念!所以返回插入元素的迭代器位置
auto it9 = g8.insert(2);
cout << "multiset插入不会失败: "<<*it9 << endl;
cout << "-----\n";
//8.删除元素 没有pop_back,pop_front这样的操作
cout <<g7.erase(1)<<endl; // 按元素的值来删除,返回删除元素的个数
for (auto& item : g8) cout << item << " "; cout << endl;
cout <<g8.erase(2)<<endl; //multiset 会把元素值是 2 的都删除了
for (auto& item : g8) cout << item << " "; cout << endl;
//erase(迭代器位置),返回删除位置后面的元素位置
auto it11 = g8.find(3);
if (it11 != g8.end())
    g8.erase(it11); //先找到3个位置,再删除。
//erase(b,e) 删除迭代器范围,返回e
g8.erase(g8.begin(), g8.end());
g7.clear(); //清空
//9.比较运算 == != < <= > >= 都有
```

```
1 2 3 4 6
1 2 2 3 4 4 6
-----
1 0
2
g7: 1找到了
lower_bound 5: 6
upper_bound 5: 6
first: 2 -- second: 3
-----
1 2 3 4 6 9 10 11
插入元素:9 是否成功:0
插入元素:8 是否成功:1
1 2 3 4 6 8 9 10 11
multiset插入不会失败: 2
-----
1
1 2 2 2 3 4 4 6
3
1 3 4 4 6
```

STL: 关联容器set/multiset (3)

```
struct A {
    A(const string& ss1 = "", int i1 = 0) :s1(ss1), d1(i1) {}
    bool operator<(const A& rhs)const { //实现了 < 没有实现 ==
        if (s1 < rhs.s1) return true;
        if (!(rhs.s1 < s1) && d1 < rhs.d1) return true;
        return false;
    }
    string s1;
    int d1;
};
```

```
abc:1 eee:3 eee:6 kkk:10
A("eee", 3) find! eee:3
kkk:10 eee:6 eee:3 abc:1
A("eee", 3) find! eee:3
```

```
//观察: set的find 是通过 < 运算符来查找的, 标准算法find是通过==
set<A> a1 = { { "abc",1 }, { "eee",6 }, { "eee",3 }, { "kkk",10 } };
for (auto& item : a1) cout << item.s1 << ":" << item.d1 << " ";
cout << endl;
auto it = a1.find(A("eee", 3));
if (it != a1.end()) {
    cout << "A(\"eee\", 3) find! " << (*it).s1 << ":" << (*it).d1 << endl;
}
//std::find(a1.begin(), a1.end(), A("eee", 3));
//出错: 提示A类型没有operator==
```

```
struct B {
    B(const string& ss1 = "", int i1 = 0) :s1(ss1), d1(i1) {}
    string s1;
    int d1;
};
```

```
class compB { //仿函数, > 返回true, 所以set从大到小
public:
    bool operator()(const B& lhs, const B& rhs)const {
        if (lhs.s1 > rhs.s1) return true;
        if (!(rhs.s1 > lhs.s1) && lhs.d1 > rhs.d1) return true;
        return false;
    }
};
```

```
set<B, compB> b1= { { "abc",1 }, { "eee",6 }, { "eee",3 }, { "kkk",10 } };
for (auto& item : b1) cout << item.s1 << ":" << item.d1 << " ";
cout << endl;
auto it0 = b1.find(B("eee", 3));
if (it0 != b1.end()) {
    cout << "A(\"eee\", 3) find! " << (*it0).s1 << ":" << (*it0).d1 << endl;
}
```

STL: 关联容器map/multimap

```
//map/multimap: 一种 key-value 的数据结构
//map(关键字不能重复), multimap(关键字可以重复) 头文件 #include <map> 红黑树
//template<class K,class V,class cmp=less<K>,class _Alloc=allocator<pair<const K,V>>>
//class map/multimap... 有4个模板参数 注意:实际存放的是 pair<const K,V>
//map与set都是同一套体系,红黑树,接口封装有区别
```

//1.初始化

```
map<string, int, less<string>> m1; //默认是less<string>
map<double, string, greater<double>> m2;
map<string, int> m3{ {"C++",80},{ "Python",90 },{"C",85} };
map<string, int> m4 = m3; //拷贝构造
map<string, int> m7(m3.begin(), m3.end()); //迭代器范围
multimap<string, int> m8 =
```

```
    { {"C++",80},{ "Python",90},{ "C",85},{ "C",83} };
for (const auto& item : m8)
    cout << item.first << ":" << item.second << " ";
cout << endl;
```

```
cout << "-----\n";
```

//2.赋值 (没有assign), swap 都与set相同

```
m3 = m4; m4 = { {"C++",80 },{ "Python",90 } };
m3.swap(m4); std::swap(m3, m4);
```

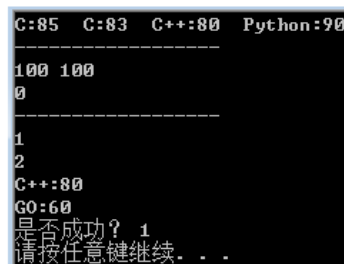
//3.容量相关,与set一致

```
m4.size(); m4.empty(); m4.max_size();
```

//4.元素访问

```
//map有下标和at访问,multimap没有(因为key可能重复,不知道哪个key)
```

```
m7["C"] = 100;
cout << m7["C"] << " " << m7.at("C") << endl;
//假如下标对应的key没有找到,则自动插入一个key,value默认初始化
cout << m7["JAVA"] << endl; // "JAVA"没有,自动插入,value=0
cout << "-----\n";
```



```
C:85 C:83 C++:80 Python:90
-----
100 100
0
-----
1
2
C++:80
GO:60
是否成功? 1
请按任意键继续...
```

//5.迭代器相关,与set一致

//6.特有算法,与set类似

```
cout << m7.count("C++") << endl; //1
cout << m8.count("C") << endl; //2
auto it1 = m7.find("C++");
if (it1 != m7.end()) { //找到
    //注意: 迭代器解引用得到的是一个 pair<const string,int>
    cout << (*it1).first << ":" << (*it1).second << endl;
    pair<const string, int> &pa1 = *it1; //注意const
}
//lower_bound, upper_bound, equal_range与set类似
```

//7.插入 insert emplace

```
//map可以通过下标直接插入,multimap不行
```

```
//直接下标相当于: auto it=m7.insert(make_pair(string("PHP"),0)).first;
// return (*it).second; //将value以引用返回
m7["PHP"] = 99; //没有"PHP"直接插入,有则修改其对应的 value
pair<map<string, int>::iterator, bool> res = m7.insert({ "GO",60 });
cout << (*res.first).first << ":" << (*res.first).second << endl;
cout << "是否成功? " << res.second << endl;
m7.insert(pair<const string, int>("GO", 60));
m7.insert(make_pair("GO", 60));
m8.emplace("CPP", 30);
m8.insert(m7.begin(), m7.end());
```

//8.删除 erase 与set一样

```
m7.erase("C");
m7.erase(m7.begin(), m7.end());
```

STL: 练习2

//练习: 将一个文件中的单词按照规则文件要求转换输出

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <map>
using namespace std;
map<string, string> build_map(ifstream &map_file);
const string& trans_word(const string& word,
                        const map<string, string> &trans_map);
```

//将转换规则文件读入map

```
map<string, string> build_map(ifstream &map_file) {
    map<string, string> trans_map;
    string key; //要替换的关键字
    string value; //替换后的内容
    char ch; //中间空格
    while (map_file >> key &&
           map_file.ignore(1),getline(map_file, value))
        trans_map[key] = value;
    return trans_map;
}
```

//给定一个单词word,在trans_map中查找是否要替换,返回转换后的string

```
const string& trans_word(const string& word,
                        const map<string, string> &trans_map) {
```

```
    auto it = trans_map.find(word);
    if (it != trans_map.end()) //找到,要替换
        return (*it).second;
    else //没找到,还是传回原来的word
        return word;
}
```

rules.txt - 记事本

文件(F) 编辑(E) 格式(O) 查

```
w who
r are
u you
y why
pic picture
thk thanks
brb be right back
```

need_trans.txt - 记事本

文件(F) 编辑(E) 格式(O) 查

```
w r u
I think u must brb
y dont u send me a pic
thk u
```

```
brb-->be right back
pic-->picture
r-->are
thk-->thanks
u-->you
w-->who
y-->why
```

```
who are you
I think you must be right back
why dont you send me a picture
thanks you
```

```
int main() {
    ifstream map_file("rules.txt"); //转换规则文件
    map<string, string> trans_map = build_map(map_file);
    for (auto& item : trans_map) //输出看下转换规则
        cout << item.first << "-->" << item.second << "\n";
    cout << endl;
    ifstream fs("need_trans.txt"); //读需要转换的文件
    string line; //每行的内容
    while (getline(fs,line)) {
        istringstream ss(line);
        string word;
        bool flag=true;
        while (ss >> word) {
            if (flag)
                flag = false;
            else
                cout << " "; //非首字
        } cout << trans_word(word, trans_map);
    } cout << endl;
} return 0;
```


STL: 无序容器(1)

set/map

```
template
    <typename Key,
        typename Comp = less<Key>,
        typename _Alloc = allocator<Key> >
class set / multiset;
```

Key类型:
有大小比较运算
默认是 <

```
template
    <typename Key,
        typename Value,
        typename Comp = less<Key>,
        typename _Alloc = allocator<pair<const Key, Value> > >
class map / multimap;
```

实现方式: **红黑树** (二叉搜索树)

1. 元素**自动排序** (通过迭代器输出是有序的)
2. 插入、查找、删除 时间复杂度: **$O(\log N)$** **稳定**
3. **内存消耗较大** (左子树指针, 右子树指针, 父结点指针, 前驱指针, 后驱指针)

unordered_set/unordered_map

```
template
    <typename Key,
        typename Hasher = hash<Key>,
        typename KeyEqual = equal_to<Key>,
        typename _Alloc = allocator<Key> >
class unordered_set / unordered_multiset;
```

Key类型:
有相等==运算
还要有个hash函数

```
template
    <typename Key,
        typename Value,
        typename Hasher = hash<Key>,
        typename KeyEqual = equal_to<Key>,
        typename _Alloc = allocator<pair<const Key, Value> > >
class unordered_map / unordered_multimap;
```

实现方式: **哈希表**

1. 元素**无序** (通过迭代器输出是无序的, 而且没有反向迭代器)
2. 插入、查找、删除 **平均时间复杂度 $O(1)$**
3. 有些极端情况下, 性能会严重下降 (比如hash函数不好)
4. **内存开销大** (通常比set/map要大一些)

STL: 无序容器 (2)

//0.set和unordered_set的比较(map和unordered_map也是一样):

```
set<int> set1 = { 3,2,1,2,4,6 };
unordered_set<int> uset1 = { 3,2,1,2,4,6 };
unordered_multiset<int> umset1 = { 3,2,1,2,4,6 };
for (auto& item : set1) cout << item << " "; cout << endl; //1 2 3 4 6 有序
for (auto& item : uset1) cout << item << " "; cout << endl; //3 2 1 4 6 无序
for (auto& item : umset1) cout << item << " "; cout << endl; //3 2 2 1 4 6 无序
//(unordered_multiset: 相等的元素, 会放在一起)
cout << "-----\n";
```

//1.初始化

```
unordered_map<string, int, hash<string>> g1; //默认就是 hash<string>
unordered_multimap<string, double> g2 = { {1,1.1},{2,2.2} };
unordered_map<string, int> g3{ { "C++",80 },{ "Python",90 },{ "C",85 } };
unordered_map<string, int> g4(g3);
unordered_map<string, int> g5 = g3; //拷贝构造
unordered_map<string, int> g7(g3.begin(), g3.end()); //迭代器范围初始化
```

//2.赋值, swap

```
g4 = g3;
unordered_multimap<string, int> g8;
g8 = { { "C++",80 },{ "Python",90 },{ "C",85 },{ "C",85 } };
g4.swap(g3); std::swap(g3, g4);
```

//3.容量相关

```
g7.size(); g7.empty(); g7.max_size();
cout << "g7中桶的数量: " << g7.bucket_count() << endl; //当前桶数量
g7.max_bucket_count(); //容器能容纳的最多的桶数量
for (int i = 0; i < g7.bucket_count(); i++) //第几个桶中元素的个数
    cout << "第" << i << "个桶中元素数量: " << g7.bucket_size(i) << endl;
cout << "\\\"C++\\\"在桶: " << g7.bucket("C++") << endl; //查找某个关键字在哪个桶中
cout << "-----\n";
```

//4.哈希策略

```
cout << "每个桶的平均元素数量(double): " << g7.load_factor() << endl;
cout << g7.max_load_factor() << endl;
//c++试图维护的平均桶大小, 需要时会添加新的桶, 使得 load_factor<=max_load_factor
g7.rehash(10); //重组存储, 使得 bucket_count(桶数) >= n
g7.reserve(100); //重组存储, 使得容器可以保存n个元素
```

//5.访问, 一样(set, map), unordered_map 有下标, at访问, multimap没有

```
g7["C++"] = 10; g7["JAVA"] = 100;
```

```
try { g7.at("C"); } catch (...) { }
```

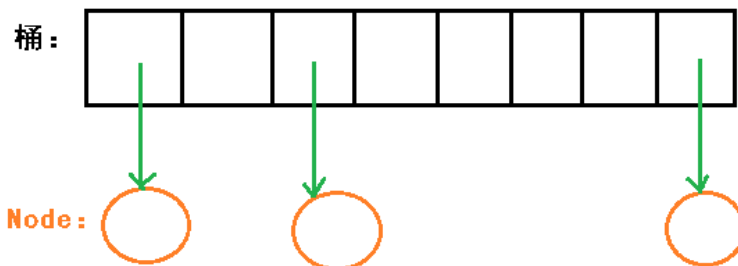
//6.迭代器相关

```
unordered_map<string, int>::iterator it01 = g7.begin();
unordered_map<string, int>::const_iterator it02 = g7.cbegin();
//没有反向迭代器, 增加了 桶迭代器 local_iterator, const_local_iterator
//表示第n个桶的首元素(begin(n), cbegin(n))和尾后迭代器(end(n), cend(n))
unordered_map<string, int>::local_iterator it03 = g7.begin(0);
```

//7.插入, 一样, insert, emplace, 包括下标插入

//8.删除(erase) 都一样

//9.特有算法一样, find, count, lower_bound, upper_bound, equal_range



```
1 2 3 4 6
3 2 1 4 6
3 2 2 1 4 6
-----
g7中桶的数量: 8
第0个桶中元素数量: 1
第1个桶中元素数量: 0
第2个桶中元素数量: 1
第3个桶中元素数量: 0
第4个桶中元素数量: 0
第5个桶中元素数量: 0
第6个桶中元素数量: 0
第7个桶中元素数量: 1
"C++"在桶: 0
-----
每个桶的平均元素数量(double): 0.375
1
请按任意键继续. . .
```

STL: 无序容器 (3)

```
class Position {
public:
    Position(int x=0,int y=0):pos_x(x),pos_y(y){}
    int get_x()const { return pos_x; }
    int get_y()const { return pos_y; }
private:
    int pos_x;
    int pos_y;
};
//作为key的Position必须要实现 == 运算符
bool operator==(const Position& lhs, const Position& rhs) {
    return (lhs.get_x() == rhs.get_x()) && (lhs.get_y() == rhs.get_y());
}
```

```
class Person {
public:
    Person(const string& n="",int a=0):name(n),age(a){}
private:
    string name;
    int age;
};
```

//创建unordered_map: key: Position --> value: Person

```
template<typename T> //boost中的一种写法
inline void hash_combine(std::size_t &seed, const T& v) {
    seed ^= std::hash<T>()(v) + 0x9e3779b9 +(seed<<6) +(seed>>2);
}
```

```
//方法1:自己定义一个hash
class MyHash {
public:
    std::size_t operator()(const Position& p)const {
        //return p.get_x() + p.get_y();
        //return hash<int>()(p.get_x()) ^ hash<int>()(p.get_y());
        std::size_t hash_result = hash<int>()(p.get_x());
        hash_combine(hash_result, p.get_y());
        return hash_result;
    }
};
```

```
//方法2:模板特例化
namespace std {
    template<> struct hash<Position> {
        std::size_t operator()(const Position& p)const {
            std::size_t hash_result = hash<int>()(p.get_x());
            hash_combine(hash_result, p.get_y());
            return hash_result;
        }
    };
}
```

```
int main() {
    //方法1:使用自己构建的 MyHash
    unordered_map<Position, Person, MyHash> g1;
    //方法2:使用hash<Position>, hash<T>的模板特例化
    unordered_map<Position, Person> g2 =
        { {Position(10,20),Person("张三",20)} };
}
```

C语言文件操作

文件打开	<code>fopen</code>	<code>FILE* fopen(char const* FileName, char const* Mode);</code>
文件关闭	<code>fclose</code>	<code>int fclose(FILE *fp);</code>
读一个字符	<code>fgetc</code>	<code>int fgetc(FILE *fp);</code>
写一个字符	<code>fputc</code>	<code>int fputc(int ch, FILE *fp);</code>
读一个字符串	<code>fgets</code>	<code>char *fgets(char *str, int n, FILE *fp);</code>
写一个字符串	<code>fputs</code>	<code>int fputs(char *str, FILE *fp);</code>
读格式化数据	<code>fscanf</code>	<code>int fscanf(FILE *fp, char *format, arg_list);</code>
写格式化数据	<code>fprintf</code>	<code>int fprintf(FILE *fp, char *format, arg_list);</code>
二进制读	<code>fread</code>	<code>int fread(void *buffer, unsigned size, unsigned count, FILE *fp);</code>
二进制写	<code>fwrite</code>	<code>int fwrite(void *buffer, unsigned size, unsigned count, FILE *fp);</code>
判断文件尾	<code>feof</code>	<code>int feof(FILE *fp);</code>
文件指针的当前位置	<code>ftell</code>	<code>long ftell(FILE *fp);</code>
随机定位文件指针	<code>fseek</code>	<code>int fseek(FILE *fp, long offset, int base);</code>
检测错误	<code>ferror</code>	<code>int ferror(FILE *fp);</code>
清除错误	<code>clearerr</code>	<code>void clearerr(FILE *fp);</code>
定位到文件头	<code>rewind</code>	<code>void rewind(FILE *fp);</code>

文件打开方式:

r 只读(文件不存在则失败)
w 只写(文件存在则先清空, 文件不存在则创建)
a 追加只写(文件不存在则创建)
r+ 读写(文件不存在则失败)
w+ 读写(文件存在则先清空, 文件不存在则创建)
a+ 读写(文件存在追加数据, 不存在则创建)

上面是以文本文件方式打开;
加**b**则以二进制格式打开
(**rb**,**wb**,**ab**,**rb+**,**wb+**,**ab+**)。

定位:

SEEK_SET: 文件开头(0),
SEEK_CUR: 当前位置(1),
SEEK_END: 文件结尾(2)

MyFstream

```
#include <cstdio>
class MyFstream {                                用C语言的文件操作 封装一个简易的MyFstream类
public:
    MyFstream(const char* path, const char* mode) { //构造, 打开文件
        fp = fopen(path, mode);
        if (!fp) throw "Open File error!";
    }
    ~MyFstream() { if (fp) fclose(fp); } //析构, 关闭文件
    MyFstream(const MyFstream&) = delete; //不能拷贝
    MyFstream& operator=(const MyFstream&) = delete; //不能赋值
    int get() { return fgetc(fp); } //读1个字符 fp --> char
    void put(int ch) { fputc(ch, fp); } //写1个字符 char --> fp
    char* getline(char* buf, int n) { return fgets(buf, n, fp); } //读一行fp->char*
    void seek(int offset, int where) { fseek(fp, offset, where); } //移动文件指针
    MyFstream & operator>> (int &val) { //从fp读一个int, fp --> int
        fscanf(fp, "%d", &val); return *this;
    }
    MyFstream & operator>> (char* val) { //从fp读1个字符串, fp --> char*
        fscanf(fp, "%s", val); return *this;
    }
    MyFstream & operator<<(const int& val) { //写一个int到文件 int --> fp
        fprintf(fp, "%d", val); return *this;
    }
    MyFstream & operator<<(const char* val) { //写一个字符串到文件 const char* --> fp
        fprintf(fp, "%s", val); return *this;
    }
private:
    FILE * fp;
};
```

```
int main() {
    MyFstream fs("1.txt", "w+");

    //写入三个整数 (像 cout << 123 类似)
    fs << 123 << " " << 234 << " " << 345 << "\n";
    //写入字符串
    fs << "I like C++\n";
    fs << "We all like coding...\n";

    int a1=0, a2=0, a3=0;
    //读整数到 a1,a2,a3 (像 cin >> a1 类似)
    fs.seek(0, SEEK_SET);
    fs >> a1 >> a2 >> a3;
    printf("%d %d %d\n", a1, a2, a3); //打印输出

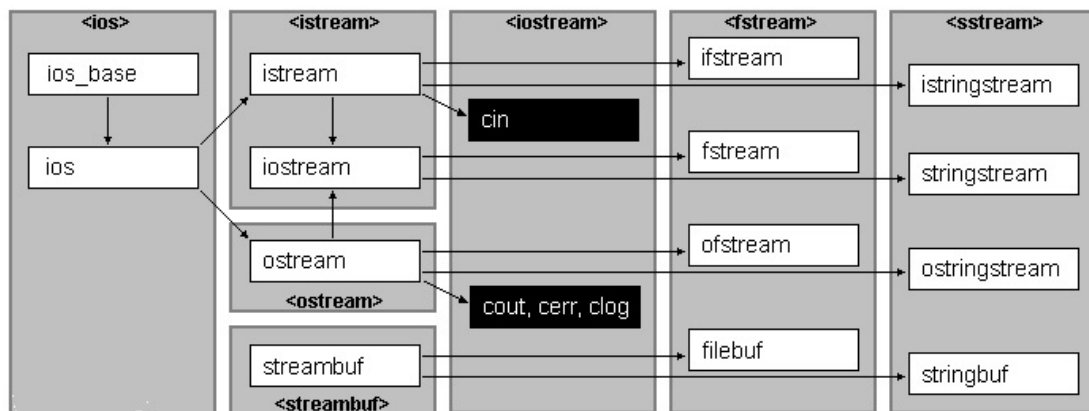
    char buf1[100],buf2[100],buf3[100];
    //读字符串到 buf1 (像 cin >> buf1 类似)
    fs >> buf1 >> buf2 >> buf3;
    printf("%s %s %s\n", buf1,buf2,buf3);
    //上面的结果, 和cin读字符串一样,遇到空格截断

    fs.get();
    //用getline读字符串到 buf1
    fs.getline(buf1, sizeof(buf1));
    printf("%s", buf1);

    return 0;
}
```

```
123 234 345
I like C++
We all like coding...
请按任意键继续. . .
```

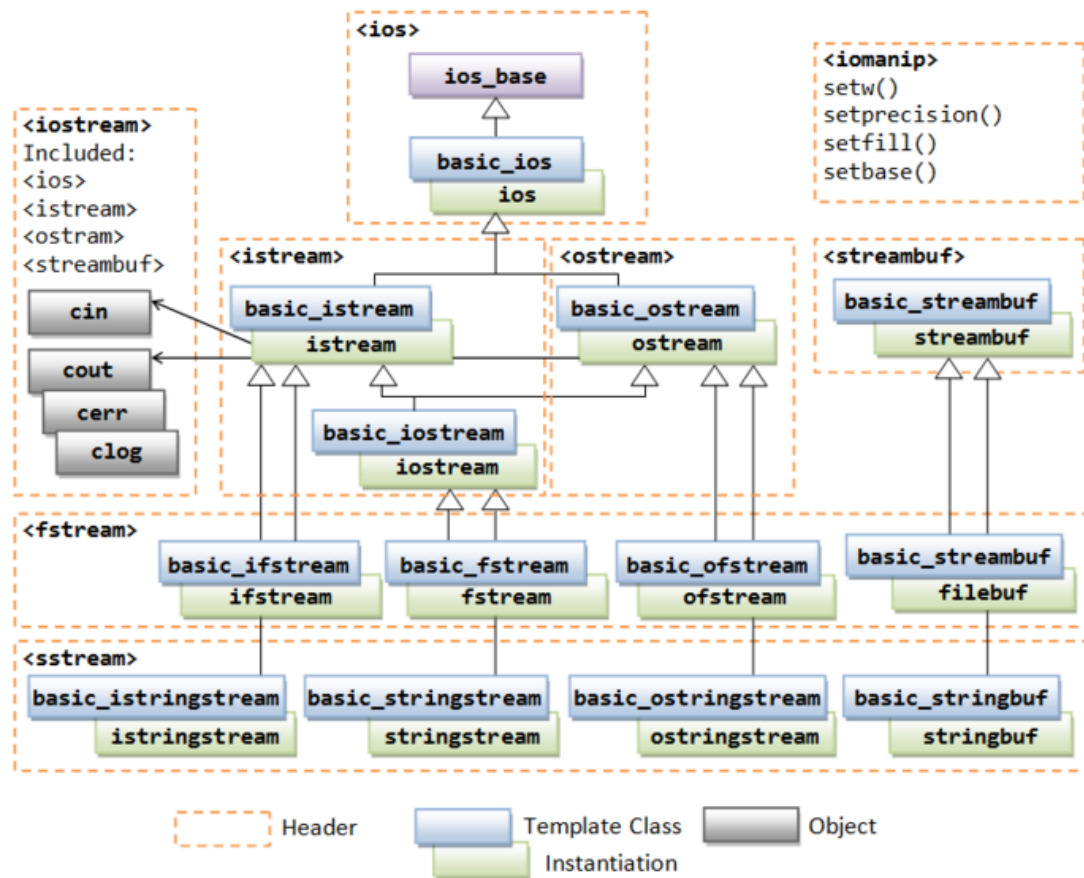
I/O流结构



C++通过一族定义在标准库中的类型来处理I/O。

类名	对象	关联
istream	cin (只读)	stdin (键盘)
ostream	cout (只写)	stdout (屏幕)
fstream	(读写)	命名文件
ifstream	(只读)	
ofstream	(只写)	
stringstream	(读写)	内存string
istringstream	(只读)	
ostringstream	(只写)	

MyFstream (读写) 命名文件



流对象不可拷贝，不可赋值。

fstream(1)

```
fstream fs1("1.txt", fstream::in | fstream::out | fstream::app);  
//读写,追加,每次写数据都写在文件的最后面  
//判断文件打开是否正常  
if (!fs1) { cout << "打开文件失败!\n"; return -1; }  
if (fs1) { cout << "文件打开ok\n"; }  
cout << "fs1文件是否打开: " << fs1.is_open() << endl;  
fs1.close(); //显式关闭,假如不显式关闭,fs1对象在析构时会自动关闭文件  
  
fstream fs2; //open打开文件  
fs2.open("2.txt", fstream::in|fstream::out|fstream::trunc);  
//读写,截断(文件存在先清空,文件不存在创建)  
if (!fs2) { cout << "open error\n"; return - 1; }  
  
//用 operator<< 输出到文件  
fs2 << 1 << " " << 2 << "\n" << "You like c++" << endl;  
//用 operator>> 从文件中读数据  
int i1, i2; string s1;  
fs2.seekg(0, fstream::beg); //移动读指针到文件头 beg cur end  
fs2 >> i1 >> i2 >> s1;  
cout << i1 << " " << i2 << " " << s1 << endl; //输出 1 2 You  
//问题: << 读数据,在读字符串的时候,遇到 空格 会截断  
//【用<< >> 输出到文件很方便,但是从文件读可能会有问题】  
cout << "1-----\n";  
  
fs2.seekg(0, fstream::beg);  
char ch;  
while ((ch = fs2.get()) != EOF) //【char get()】 读入一个字符并返回  
    cout.put(ch);  
fs2.clear(); //状态位复位  
fs2.seekg(0, fstream::beg);  
while (fs2.get(ch)) //【fstream& get(char&)】  
    cout.put(ch);  
cout << "2-----\n";
```

```
文件打开ok  
fs1文件是否打开: 1  
1 2 You  
1-----  
1 2  
You like c++  
1 2  
You like c++  
2-----  
1  
1 2  
You like c++  
3-----  
1 2  
You like c++  
4-----  
1 2  
You like c++  
5-----
```

```
fs2.clear(); //状态位复位  
fs2.seekg(0, fstream::beg);  
char buf[1024];  
fs2 >> buf; cout << buf << endl; //读字符串,遇到空格截断  
fs2.seekg(0, fstream::beg);  
  
//【fstream& get(char*,int,char)】 字符数组/指针,个数,终止字符  
//但是,该成员函数不会将 终止字符取走,所以后面加ignore(1),忽略1个字符  
while (fs2.get(buf, 1024, '\n')) {  
    cout << buf << endl;  
    fs2.ignore(1);  
}  
cout << "3-----\n";  
fs2.clear();  
fs2.seekg(0, fstream::beg);  
//【istream& getline(char*, int, char)】 字符数组/指针,个数,终止字符  
//但是,该成员函数不会将终止字符存入char*,但是流中越过了截止符  
while (fs2.getline(buf, 1024, '\n'))  
    cout << buf << endl;  
cout << "4-----\n";  
  
fs2.clear();  
fs2.seekg(0, fstream::beg);  
//【函数模板 不是成员函数!】  
// stream& getline(stream&, string&)  
string tmps;  
while (getline(fs2, tmps))  
    cout << tmps << endl;  
cout << "5-----\n";
```

in	为输入(读)而打开文件
out	为输出(写)而打开文件
ate	初始位置: 文件尾
app	所有输出附加在文件末尾
trunc	如果文件已存在则先清空文件
binary	二进制方式

fstream(2)

```
fs2.clear();
fs2.seekg(0, fstream::beg);
fs2.ignore(100, 'Y'); //最多忽略100个字符,或者遇到'Y','Y'字符会被忽略掉!
fs2.putback('Y'); //重新将'Y'放回流中
cout << "peek:" << (char)fs2.peek() << endl; //peek()窥视,但是不取数据
getline(fs2, tmps);
cout << tmps << endl;
fs2.close();
cout << "6-----\n";

ifstream ifs("2.txt",ifstream::in | ifstream::binary);
//ifstream默认打开方式: in
ofstream ofs("1.txt", ofstream::out | ofstream::trunc | ofstream::binary);
//ofstream默认打开方式: out | trunc
ifs.read(buf, 1024); //二进制read, 最多读1024字节
int cnt = ifs.gcount(); //上次读取的字节数
ofs.write(buf, cnt); //将buf写入流,字节数为 cnt

struct Student { char name[20]; int age; char sex; };
Student stu[] = { {"张三",10,'m'}, {"李四",20,'f'}, {"王五",30,'m'} };
fstream fs_stu("stu.data",fstream::in|fstream::out|fstream::trunc|fstream::binary);
fs_stu.write(reinterpret_cast<char*>(stu), sizeof(stu));
fs_stu.seekg(0, fstream::beg);
Student tmp_stu;
for (int i = 0; i < 3; i++) {
    fs_stu.read(reinterpret_cast<char*>(&tmp_stu), sizeof(tmp_stu));
    cout << tmp_stu.name << ":" << tmp_stu.age << ":" << tmp_stu.sex << endl;
}
cout << "7-----\n";
```

```
6
张三:10:m
李四:20:f
王五:30:m
7
fs.eof(): 0
fs.fail(): 0
fs.bad(): 0
fs.good(): 1
7.1-----
1 2
You like c++
fs.eof(): 1
fs.fail(): 1
fs.bad(): 0
fs.good(): 0
7.2-----
fs.eof(): 0
fs.fail(): 0
fs.bad(): 0
fs.good(): 1
7.3-----
fs.eof(): 0
fs.fail(): 1
fs.bad(): 0
fs.good(): 0
7.4-----
```

```
ifstream fs("2.txt");
if (!fs) { cout << "open error\n"; return -1; }
//流状态判断:
//fs.eof() 到达尾部返回true, 对于cin就是输入ctrl+z
//fs.fail() 上次IO操作失败返回true
//fs.bad() 返回true表示流崩溃
//fs.good() 流处于有效状态返回true
cout << "fs.eof(): " << fs.eof() << endl //0
    << "fs.fail(): " << fs.fail() << endl //0
    << "fs.bad(): " << fs.bad() << endl //0
    << "fs.good(): " << fs.good() << endl;//1
cout << "7.1-----\n";
while (fs.get(ch)) cout << ch;
//当前位置在文件尾部
cout << "fs.eof(): " << fs.eof() << endl //1
    << "fs.fail(): " << fs.fail() << endl //1
    << "fs.bad(): " << fs.bad() << endl //0
    << "fs.good(): " << fs.good() << endl;//0
cout << "7.2-----\n";
fs.clear(); //状态复位,流有效
fs.seekg(0, ifstream::beg);
cout << "fs.eof(): " << fs.eof() << endl //0
    << "fs.fail(): " << fs.fail() << endl //0
    << "fs.bad(): " << fs.bad() << endl //0
    << "fs.good(): " << fs.good() << endl;//1
cout << "7.3-----\n";
int a1, a2, a3;
fs >> a1 >> a2 >> a3; //a3会错,要求读入int,但是没有
cout << "fs.eof(): " << fs.eof() << endl //0
    << "fs.fail(): " << fs.fail() << endl //1
    << "fs.bad(): " << fs.bad() << endl //0
    << "fs.good(): " << fs.good() << endl;//0
cout << "7.4-----\n";
```

stringstream

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

int main() {
    //istringstream 从string中读取数据
    //ostringstream 向string写入数据
    //stringstream 既可以读,也可以写
    //基本操作和前面的类似(继承体系类似)

    string str1("a:111111\nb:222222\nc:333333\n");
    stringstream ss1;
    stringstream ss2(str1); //保存str1的拷贝

    ss1 << 1 << " " << 2 << " I like C++" << endl;
    int i1, i2;
    string tmps;
    ss1 >> i1 >> i2 >> tmps;
    cout <<i1<<" "<<i2<<" "<<tmps<<endl; //与fstream一样的效果
    ss1.seekg(0, stringstream::beg);
    getline(ss1, tmps); //ok,字符串没有截断
    cout << tmps << endl;
    cout << "1-----\n";
```

```
1 2 I
1 2 I like C++
1-----
1 2 I like C++
2-----
a:111111
b:222222
c:333333
3-----
张三:10:m
李四:20:f
王五:0:m
end-----
请按任意键继续.
```

```
//将string流中的内容 复制到 string对象
string str2 = ss1.str(); //ss1流中的数据 拷贝到了str2
cout << str2;
cout << "2-----\n";
//改变string流中的内容
ss1.str(str1); //用str1的内容拷贝到 ss1流中
ss1.seekg(0, stringstream::beg);
while(getline(ss1, tmps))
    cout << tmps << endl;
cout << "3-----\n";
//ss1.clear() 不是清空流数据,而是流状态复位
//要清空数据,可以这样写:
ss1.str("");

//二进制的read write也可以
struct Student { char name[20]; int age; char sex; };
Student stu[] = { { "张三",10,'m' },{ "李四",20,'f' },{ "王五",0,'m' } };

ostringstream os1;
os1.write((char*)stu, sizeof(stu));
istringstream is1(os1.str());
Student tmp;
for (int i = 0; i < 3; ++i) {
    is1.read((char*)&tmp, sizeof(tmp));
    cout << tmp.name << ":" << tmp.age << ":" << tmp.sex << endl;
}
```

缓冲区

输出设备或磁盘文件，通常读写速度要比内存慢非常多，假如每次写操作都要直接操作文件，对系统性能会有严重影响，通常会开辟一块内存作为输出缓冲区(buffer)，写操作只是写到缓冲区中，等多次或者需要时将缓冲区数据真正写入文件中。如此，来解决高速设备和低速设备的匹配。

缓冲机制：操作系统可以将程序的多个输出操作合并成一个系统级写操作，从而大幅度提升性能。

缓冲刷新：数据真正写到输出设备或文件。刷新原因：

1. 程序正常结束，缓冲刷新；
2. 缓冲区满，刷新缓冲，后续新数据才能写入缓冲区；
3. 使用操作符如endl来显示刷新缓冲区； `cout << "hi" << endl; cout << "hi" << flush; cout << "hi" << ends;`
4. 使用操作符unitbuf设置流状态； `cout << unitbuf;` 此后所有cout操作都立即刷新； `cout << nunitbuf;` 此后的cout操作恢复原样，回到正常的缓冲方式。
5. 一个输出流可被关联到另一个流。当读写被关联的流时，该流缓冲区刷新。默认情况下，cin和cerr都关联到cout. 因此读cin和写cerr都会导致cout的缓冲区刷新。

注意：程序崩溃，输出缓冲区不会被刷新。(调试程序时要注意！)

练习:文本查询程序(1)

文本查询程序: 在一个给定的文件中查询单词。
查询结果以及输出: 显示所有出现该单词的行号和行内容。如果一个单词在一行中多次出现, 那么该行只显示一次。

map, set, vector 和 shared_ptr 的结合使用。

分析:

当程序读取输入文件时, 分解出所有的单词, 并且要记录下每个单词出现的行号。

(1) 用一个 `vector<string>` 来保存文件内容, 每一行作为一个元素存入, 按次序存时, 实际上行号就是 `vector` 中元素的位置。

(2) 根据单词来搜索它出现的所有行号, 可以考虑 `map<...>`, 其中 `key` 可以用单词, `value` 可以用 `vector` 来保存所有的行号, 但是, 题目要求不能有重复行号, 所以 `value` 用 `set` 更合适。那么就是 `map<string, set<int>>`, `set<int>` 集合中存放了不重复的行号

有了上面这些数据, 在查询输出的时候, 只要在 `map` 中找到该单词 (找不到直接输出没有找到即可), 就可以得到一个 `set<int>`, 然后遍历该 `set` 中的元素, 得到行号, 作为下标值在上面那个 `vector<string>` 中得到该行的内容, 显示即可。

`vector<string>` 每个元素都是一行内容

文件第1行内容	第2行	第n行
---------	-----	-------	-----

文件内容:

```
All things come to those who wait
Victory won't come to me unless I go to it
We must accept finite disappointment, but we must never lose
A thousand-li journey is started by taking the first step.
Never, never, never, never give up
A man is not old as long as he is seeking something.
While there is life there is hope
I am a slow walker, but I never walk backwards.
```

程序运行的输出结果:

```
输入要查找的单词 <输入q退出>: a
a 共在 3 行中出现:
<4> A thousand-li journey is started by taking the first step.
<6> A man is not old as long as he is seeking something.
<8> I am a slow walker, but I never walk backwards.

输入要查找的单词 <输入q退出>: first
first 共在 1 行中出现:
<4> A thousand-li journey is started by taking the first step.

输入要查找的单词 <输入q退出>: never
never 共在 3 行中出现:
<3> We must accept finite disappointment, but we must never lose
<5> Never, never, never, never give up
<8> I am a slow walker, but I never walk backwards.

输入要查找的单词 <输入q退出>: _
```

`map<string, set<int>>` 单词 到 行号集合 的映射表

key: 单词 "a"	value: { 3, 5, 7 } (出现的行号)
key: 单词 "first"	value: { 3 } (第3行出现)
key: 单词 "never"	value: { 2, 4, 7 } 是个set
.....

练习:文本查询程序(2)

```
class TextQuery {
public:
    //根据文件生成相关需要的数据:按行保存文件到vector,建立单词到行号集合的映射
    TextQuery(istream& ifs):file(make_shared<vector<string>>()) { //根据文件,构造
        string text;
        while (getline(ifs, text)) { //一行一行读到 text中
            (*file).push_back(text); //存入vector
            int n = (*file).size() - 1; //当前行号
            istringstream line(text); //准备读该行的每个单词
            string word;
            while (line >> word) { //该行中的每个单词,存入map(wm中) //转小写
                transform(word.begin(), word.end(), word.begin(), ::tolower);
                shared_ptr<set<int>> &lines = wm[word]; //没有该单词,也会自动插入,
                if (!lines) //但是此时value是空指针
                    lines.reset(new set<int>); //创建一个新的 set<int>
                (*lines).insert(n); //将此行号插入 set中
            }
        }
    }
    void show()const {
        int i = 1;
        for (const auto& item : *file) //显示vector:
            cout << "(" << i++ << " " << item << endl;
        for (const auto& item : wm) { //显示map:
            cout << item.first << "-->";
            for (const auto& item1 : *item.second) cout << item1<<" ";cout<<endl;
        }
    }
private:
    shared_ptr<vector<string>> file; //文件内容,每个元素都是一行,并且下标和行号对应
    map<string, shared_ptr<set<int>>> wm; //单词 --> 保存行号的set 的映射
};
```

构建一个类: **TextQuery**

数据成员: 1. 以每行文件内容为元素的vector<string>
暂且先以**智能指针**存放 shared_ptr<vector<string>> file

数据成员: 2. map<string, shared_ptr<set<int>>> wm
暂且集合set<int>也以**智能指针**的形式存放

构造函数: 传入一个文件流, 读取每行内容存入vector, 解析每个单词, 将其出现的行号放入set中, 然后存入map

show函数: 测试用, 用来观察构建的vector和map是否正确

测试程序:

```
ifstream ifs("word_query.txt");
if (!ifs){cout << "open error!\n";return -1;}
TextQuery tq(ifs);
tq.show();
```

```
<1> All things come to those who wait
<2> Victory won't come to me unless I go to it
<3> We must accept finite disappointment, but we must never lose
<4> A thousand-li journey is started by taking the first step.
<5> Never, never, never, never give up
<6> A man is not old as long as he is seeking something.
<7> While there is life there is hope
<8> I am a slow walker, but I never walk backwards.
a-->3 5 7
accept-->2
all-->0
am-->7
as-->5
```

练习:文本查询程序(3)

```
class QueryResult {
    friend ostream& print(ostream& out, const QueryResult &qr);
public:
    QueryResult(const string& s,
               const shared_ptr<vector<string>>& f,
               const shared_ptr<set<int>> &p):
        sought(s),file(f),lines(p){}
private:
    string sought; //查询的单词
    shared_ptr<vector<string>> file; //文件内容
    shared_ptr<set<int>> lines; //行号集合
};

ostream& print(ostream& out, const QueryResult &qr) {
    out << qr.sought << " 共有 " << (*qr.lines).size() << " 行中出现:\n";
    for (const auto& item : *qr.lines)
        cout << " (" << item + 1 << ") " << (*qr.file)[item] << endl;
    return out;
}
```

构建一个类: **QueryResult**

该类用来保存某个单词的查询结果: 查询结果包括: 查询的单词, 该单词出现过的所有行号, 包括这些行的内容。

这里通过智能指针shared_ptr实际上共享了 TextQuery类对象的数据, 而不是再复制一份。

非成员函数print负责将QueryResult类对象输出到流out。实际上, 通过传入参数out的不同, 可以输出到屏幕, 也可以输出到文件等等。

练习:文本查询程序(4)

```
//查询函数,输入单词,返回一个QueryResult对象
QueryResult TextQuery::query(const string& sought)const {
    //如果没有找到该单词,则返回下面这个 空set<int>
    static shared_ptr<set<int>> nodata = make_shared<set<int>>();
    auto it = wm.find(sought);
    if (it != wm.end()) //找到了
        return{ sought,file, (*it).second };
    else
        return{ sought,file,nodata };
}

int main() {
    ifstream ifs("word_query.txt");
    if (!ifs) { cout << "open error!\n"; return -1; }
    TextQuery tq(ifs);
    tq.show();
    cout << "-----\n";
    while (true) {
        cout << "\n输入要查找的单词 (输入q退出): ";
        string s;
        if (cin >> s && s != "q") { //转小写
            transform(s.begin(), s.end(),s.begin(),::tolower);
            print(cout, tq.query(s));
        }
        else
            break;
    }
}
```

类TextQuery增加一个成员函数用来实现查询 **query(单词)**: 通过map的find返回的迭代器是否map的end来判断是否找到, 没有找到则利用一个static创建的空set, 找到则将对应的set拿出, 然后**构建一个QueryResult临时对象返回**。

main()函数中示例了如何调用TextQuery和QueryResult类对象来完成查询工作。

文本查询程序深入: 在C++ primer第5版562页有更加深入的探讨, 要求能实现混合运算, 比如 单词1 & 单词2 | 单词3 这样的操作, 可自行研习。

```
输入要查找的单词 <输入q退出>: never
never 共在 3 行中出现:
(3) We must accept finite disappointment, but we must never lose
(5) Never, never, never, never give up
(8) I am a slow walker, but I never walk backwards.

输入要查找的单词 <输入q退出>: test
test 共在 0 行中出现:

输入要查找的单词 <输入q退出>: who
who 共在 1 行中出现:
(1) All things come to those who wait

输入要查找的单词 <输入q退出>: q
请按任意键继续. . .
```