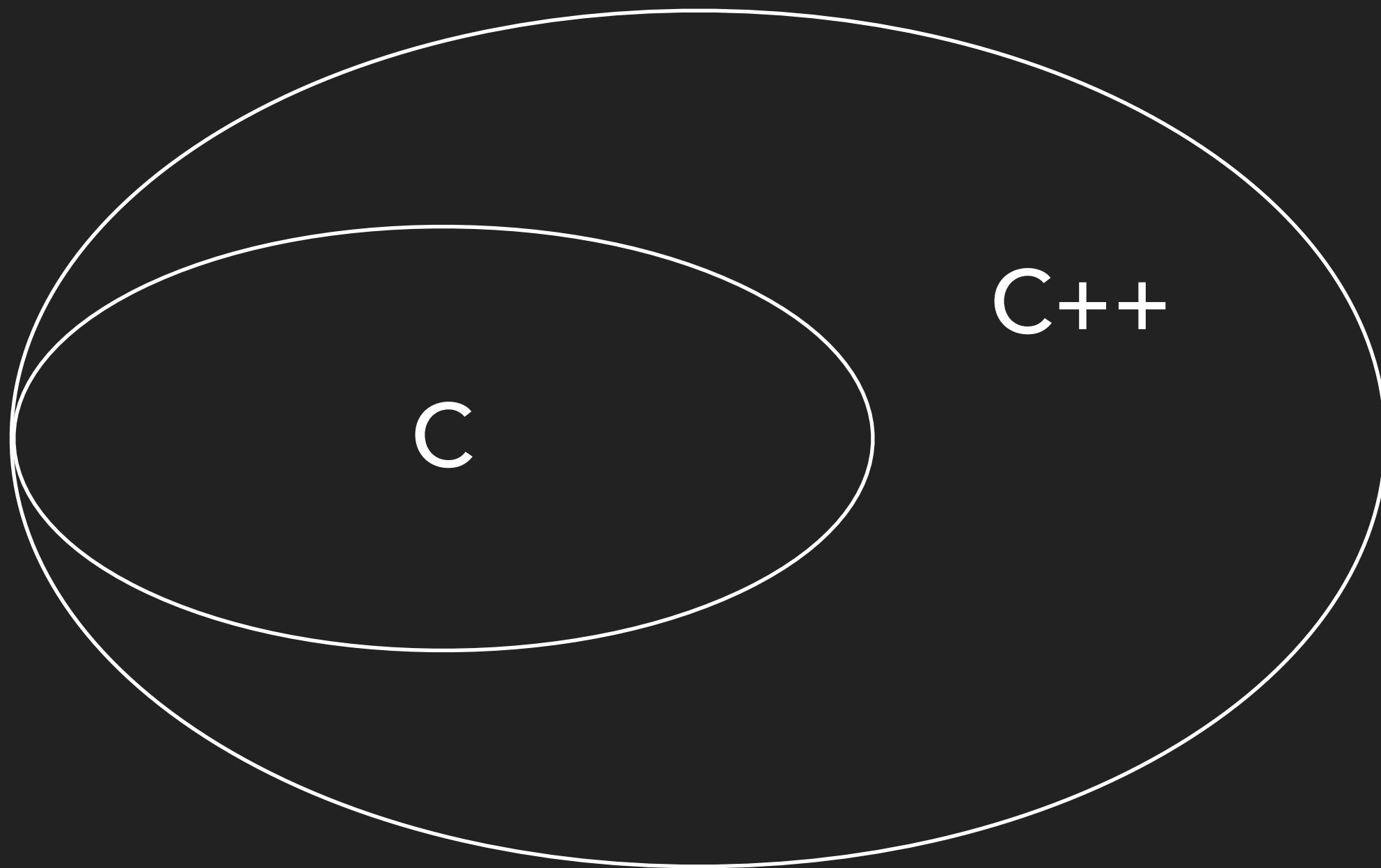


THE INTRODUCTION TO

---

C++ & STL



# a + b problem (C ver.)

```
#include <stdio.h>
```

```
int main() {  
    int a, b;  
    scanf("%d%d", &a, &b);  
    printf("%d", a + b);  
    return 0;  
}
```

# a + b problem (C++ ver.)

```
#include <cstdio>
```

```
int main() {  
    int a, b;  
    scanf("%d%d", &a, &b);  
    printf("%d", a + b);  
    return 0;  
}
```

# a + b problem (C++ ver.)

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

```
int main() {
    int a, b;
    cin >> a >> b;
    cout << a + b;
    return 0;
}
```

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

```
int main() {
    int a, b;
    cin >> a >> b;
    cout << a + b;
    return 0;
}
```

iostream 是 C++ 特有的头文件，提供输入输出流

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

```
int main() {
    int a, b;
    cin >> a >> b;
    cout << a + b;
    return 0;
}
```

namespace 是 C++  
中用来处理函数重名  
的问题

```
#include <iostream>
```

```
int main() {  
    int a, b;  
    std::cin >> a >> b;  
    std::cout << a + b;  
    return 0;  
}
```

如果不用 using  
namespace std 的  
话，需要手动在 std  
的函数前面加上  
std::



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

```
int main() {
    int a, b;
    cin >> a >> b;
    cout << a + b;
    return 0;
}
```

cin 相当于 C 语言中的 scanf()

优势是不需要写 %d 之类的东西就可以直接对变量赋值

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

```
int main() {
    int a, b;
    cin >> a >> b;
    cout << a + b;
    return 0;
}
```

cout 相当于 C 语言中的  
printf()

优势与 cin 相同

# reference

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

```
void swap2(int &a, int &b) {
    int c = a;
    a = b;
    b = c;
}
```

```
int main() {
    int a, b;
    cin >> a >> b;
    swap2(a, b);
    cout << a << " " << b;
    return 0;
}
```

input:

1 2

output:

2 1

# Point struct (C ver.)

```
#include <stdio.h>
```

```
struct Point {  
    double x, y;  
};
```

```
void printPoint(struct Point p) {  
    printf("%lf %lf", p.x, p.y);  
}
```

```
int main() {  
    struct Point p;  
    p.x = 1; p.y = 2;  
    printPoint(p);  
    return 0;  
}
```

# Point struct (C++ ver.)

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

struct Point {
    double x, y;

    Point(double x, double y):
        x(x), y(y) {}

    void print() {
        printf("%lf %lf", x, y);
    }
};

int main() {
    Point p = Point(1, 2);
    p.print();
    return 0;
}
```

```
struct Point {  
    double x, y;
```

```
    Point(double x, double y):  
        x(x), y(y) {}
```

```
    void print() {  
        printf("%lf %lf", x, y);  
    }
```

```
};
```

```
int main() {  
    Point p = Point(1, 2);  
    p.print();  
    return 0;  
}
```

Point() 是构造函数  
(constructor)

用于初始化结构体

```
Point(double x, double y) {  
    this->x = x;  
    this->y = y;  
}
```

x(x) y(y) 是简单  
写法，也可以写成上  
面这种形式

```
Point(double x = 0, double y = 0) {  
    this->x = x;  
    this->y = y;  
}
```

对 constructor  
的参数直接赋值作为  
x 和 y 的默认值



```
struct Point {  
    double x, y;  
  
    Point(double x, double y):  
        x(x), y(y) {}  
  
    void print() {  
        printf("%lf %lf", x, y);  
    }  
};  
  
int main() {  
    Point p = Point(1, 2);  
    p.print();  
    return 0;  
}
```

C++ 中的 struct 可以  
写函数，调用方法和  
成员变量相同

# overloading operators

```
struct Point {  
    int x, y;  
    .....  
};
```

```
Point operator + (const Point &lhs, const Point &rhs) {  
    return Point(lhs.x + rhs.x, lhs.y + rhs.y);  
}
```

```
int main() {  
    Point a(1, 2), b(3, 4);  
    Point c = a + b;  
    c.print();  
    return 0;  
}
```

# overloading operators

```
Point operator + (const Point &lhs, const Point &rhs) {  
    return Point(lhs.x + rhs.x, lhs.y + rhs.y);  
}
```

返回值类型 `operator` 需要重载的运算符 (

```
    const 类型1 &lhs,  
    const 类型2 &rhs  
) {  
    .....  
    return .....;  
}
```

(单目运算符只需要有一个参数)

# overloading operators

```
struct Point {  
    int x, y;
```

```
    .....  
};
```

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

函数结尾处的 `const` 表示这个函数  
不会改变成员变量的值

# template

```
int sum(int *begin, int *end) {  
    int sum = 0;  
    for (int *i = begin; i != end; i++) {  
        sum += *i;  
    }  
    return sum;  
}
```

```
int main() {  
    int a[4] = {1, 3, 4, 5};  
    cout << sum(a, a + 4);  
    // 如果我还需要计算 double 数组的序列和应该怎么办?  
    return 0;  
}
```

# template

```
int sum(int *begin, int *end) {  
    int sum = 0;  
    for (int *i = begin; i != end; i++) {  
        sum += *i;  
    }  
    return sum;  
}
```

```
double sum(double *begin, double *end) {  
    double sum = 0;  
    for (double *i = begin; i != end; i++) {  
        sum += *i;  
    }  
    return sum;  
}
```

# template

```
template <typename T>
T sum(T *begin, T *last) {
    T sum = 0;
    for (T *i = begin; i != last; i++) {
        sum += *i;
    }
    return sum;
}
```

output:

13

```
int main() {
    int a[4] = {1, 3, 4, 5};
    double b[4] = {1, 3, 4, 5.5};
    cout << sum(a, a + 4) << endl;
    cout << sum(b, b + 4) << endl;
    return 0;
}
```

13.5

# template

```
template <typename T>
struct Point {
    T x, y;
```

```
    Point(T x = 0, T y = 0):
        x(x), y(y) {}
};
```

```
int main() {
    Point<int> p1(1, 2);
    Point<double> p2(1.2, 2.3);
    return 0;
}
```



---

## 关于 STL（标准模板库）

- ▶ Standard Template Library, 缩写: STL
- ▶ STL 是一个 C++ 软件库, 里面包含算法 (algorithms)、容器 (containers)、函数 (functions)、迭代器 (iterators)
- ▶ 我这里重点讲解的是 containers, iterators, algorithms

# 字符串 (string)

```
#include <string>
```

```
string str = "C++ string";
```

```
int main() {  
    string a, b;  
    cin >> a >> b;  
    cout << a + b;  
    return 0;  
}
```

input:

**abcd**

**edgh**

output:

**abcdefgh**

```
int main() {  
    string str;  
    getline(cin, str);  
    cout << str;  
    return 0;  
}
```

input:

**abcd edgh**

output:

**abcd efgh**

```
int main() {  
    string a = "abcdefgh";  
    cout << a.length() << endl;  
    for (int i = 0; i < a.length(); i++) {  
        printf("%c", a[i]);  
    }  
    return 0;  
}
```

output:

8

**abcdefgh**

## 获取子串方法

---

```
str.substr(int idx, int len);
```

- ▶ idx: 子串开始的位置
- ▶ len: 子串总共的长度

```
string str = "abcdefgh";  
cout << str.substr(1, 2);  
// 输出为 bc
```

```
cout << str.substr(1);  
// 输出为 bcdefgh
```

## 可变长数组 (vector)

```
#include <vector>
```

```
vector<Type> v;
```

## 可变长数组 (vector)

```
#include <vector>
```

```
vector<int> v;
```



## 可变长数组 (vector)

```
#include <vector>
```

```
vector<string> v;
```

## 可变长数组 (vector)

```
#include <vector>
```

```
vector<Point> v;
```

---

## vector 使用方法

- ▶ `v.push_back(item)`: 向 `v` 后面添加一个元素  $O(1)$
- ▶ `v.pop_back()`: 删除 `v` 最后一个元素  $O(1)$
- ▶ `v.size()`: 获取 `v` 中元素个数  $O(1)$
- ▶ `v.resize(n)`: 把 `v` 的长度设定为 `n` 个元素  $O(|n - \text{size}|)$
- ▶ `v.empty()`: 判断 `v` 是否为空  $O(1)$
- ▶ `v.clear()`: 清空 `v` 中的元素  $O(\text{size})$
- ▶ `v[index]`: 获取 `v` 中第 `index` 个元素  $O(1)$

---

## vector 使用方法举例

```
vector<int> v;
```

```
for (int i = 0; i < 4; i++) {  
    v.push_back(i);  
}
```

```
cout << v.empty() << endl; // 0  
cout << v.size() << endl;  // 4  
cout << v[2] << endl;      // 2
```

## 关于 iterator (迭代器)

- ▶ eg. `vector<int>::iterator it = v.begin();`
- ▶ 在这里 `it` 类似于一个指针，指向 `v` 的第一个元素，`*it` 是第一个元素的值
- ▶ `it` 也可以进行加减操作，例如 `it + 3` 指向第四个元素，因此我们可以用 `it` 遍历整个 `vector`:

```
for (vector<int>::iterator it = v.begin();  
     it != v.end(); it++) {  
    cout << *it << endl;  
}
```

---

`v.insert(position, item)`

- ▶ insert 的功能是在 v 中某个位置插入一个元素
- ▶ position 是一个 iterator, 表示要插入的位置
- ▶ item 是需要插入的元素
- ▶ 时间复杂度为  $O(\text{size} - \text{pos})$

```
vector<int>::iterator it = v.begin() + 3;  
v.insert(it, 1000);
```

---

`v.erase(position)`

- ▶ `erase` 的功能是删除 `v` 中某个位置的元素
- ▶ 和 `insert` 一样, `position` 是一个 `iterator`, 表示需要删除的位置, 例如:
- ▶ 时间复杂度为  $O(\text{size} - \text{pos})$

```
vector<int>::iterator it = v.begin() + 3;  
v.erase(it);
```

# 队列 (queue)

```
#include <queue>
```

```
queue<Type> q;
```



---

## queue 使用方法

- ▶ `q.push(item)`: 在 `q` 的最后添加一个元素  $O(1)$
- ▶ `q.pop()`: 使 `q` 最前面的元素出队  $O(1)$
- ▶ `q.front()`: 获取 `q` 最前面的元素  $O(1)$
- ▶ `q.back()`: 获取 `q` 最后面的元素  $O(1)$
- ▶ `q.size()`: 获取 `q` 中元素个数  $O(1)$
- ▶ `q.empty()`: 判断 `q` 是否为空  $O(1)$

## 优先队列 (priority\_queue)

```
#include <queue>
```

```
priority_queue<Type> pq;
```

## 关于 priority\_queue

- ▶ pq 中出队顺序与插入顺序无关，与数据优先级有关
- ▶ pq 中的元素的类型必须定义过「小于」运算符，因此想把 struct 放到 pq 中的话**必须手动重载「<」运算符**
- ▶ 例如：priority\_queue<int> pq 中永远是数字最大的先出队

```
priority_queue<int> pq;  
pq.push(1);  
pq.push(3);  
pq.push(2);  
while (!pq.empty()) {  
    cout << pq.top() << endl;  
    pq.pop();  
}
```

output:

3

2

1

---

## priority\_queue 使用方法

- ▶ `pq.push(item)`: 在 pq 中添加一个元素  $O(\log n)$
- ▶ `pq.pop()`: 使 pq 最前面的元素出队  $O(\log n)$
- ▶ `pq.top()`: 获取 pq 最前面的元素  $O(1)$
- ▶ `pq.size()`: 获取 pq 中元素个数  $O(1)$
- ▶ `pq.empty()`: 判断 pq 是否为空  $O(1)$

## 洛谷 P3378 【模板】堆 (1000ms / 128MB)

### 输入：

第一行包含一个整数N，表示操作的个数。接下来N行，每行包含1个或2个正整数，表示三种操作，格式如下：

操作1： 1 x (表示将x插入到堆中)

操作2： 2 (输出该小根堆内的最小数)

操作3： 3 (删除该小根堆内的最小数)

$N \leq 1000000$

### 输出：

包含若干行正整数，每行依次对应一个操作2的结果。

`priority_queue<Type, Container, Functional>`

- ▶ `Type`: 数据类型
- ▶ `Container`: 盛放数据的容器, 默认用的是 `vector`, 也可以用 `queue`
- ▶ `Functional`: 元素之间的比较方法

## 实现每次让最小值出队的方法

```
// 方法一：将第三个参数设定为 greater<int>  
priority_queue<int, vector<int>, greater<int> > pq;
```

```
pq.push(1);  
pq.push(3);  
pq.push(2);  
while (!pq.empty()) {  
    cout << pq.top() << endl;  
    pq.pop();  
}
```

output:

1


2

3

## 实现每次让最小值出队的方法

```
// 方法一：将第三个参数设定为 greater<int>  
priority_queue<int, vector<int>, greater<int> > pq;
```

```
pq.push(1);  
pq.push(3);  
pq.push(2);  
while (!pq.empty()) {  
    cout << pq.top() << endl;  
    pq.pop();  
}
```



注意最后这两个「>」  
不要写在一起，因为这  
有可能会被认为是  
「>>」运算符



---

// 方法二：写一个 struct 并在其内部重载「()」运算符，  
// 然后将这个 struct 作为第三个参数

```
struct comp {  
    bool operator() (const int &lhs, const int &rhs) const {  
        return lhs > rhs;  
    }  
};
```

```
priority_queue<int, vector<int>, comp > pq;
```

output:

```
pq.push(1);  
pq.push(3);  
pq.push(2);  
while (!pq.empty()) {  
    cout << pq.top() << endl;  
    pq.pop();  
}
```

1

2

3

## 双端队列 (deque)

```
#include <deque>
```

```
deque<Type> dq;
```

## 栈 (stack)

```
#include <stack>
```

```
stack<Type> s;
```

## 集合 (set)

```
#include <set>
```

```
set<Type> s;
```

---

## 关于 set

- ▶ set 是按照特定顺序存储元素的容器
- ▶ set 中所有元素只能出现一次
- ▶ 因为 set 中元素是有序的，所以存储的元素必须已经定义过「小于」运算符（因此如果想在 set 中存放 struct 的话必须手动重载「<」运算符）
- ▶ 另外还有支持多个元素具有相同值的 multiset

---

## set 使用方法 (1)

- ▶ `s.insert(item)`: 在 `s` 中插入一个元素  $O(\log n)$
- ▶ `s.size()`: 获取 `s` 中元素个数  $O(1)$
- ▶ `s.empty()`: 判断 `s` 是否为空  $O(1)$
- ▶ `s.clear()`: 清空 `s`  $O(n)$
- ▶ `s.find(item)`: 在 `s` 中查找一个元素并返回其 iterator, 找不到的话返回 `s.end()`  $O(\log n)$

---

## set 使用方法 (2)

- ▶ `s.count(item)`: 返回 `s` 中 `item` 的数量。因为所有元素只能在 `s` 中出现一次, 所以返回值只能是 0 或 1  $O(\log n)$
- ▶ `s.erase(position)`: 删除 `s` 中对应位置的元素 (均摊时间为常数)
- ▶ `s.erase(item)`: 删除 `s` 中对应元素  $O(\log n)$
- ▶ `s.erase(pos1, pos2)`: 删除 `[pos1, pos2)` 这个区间的位置的元素  $O(pos2 - pos1)$

---

## set 使用方法 (3)

- ▶ 因为 set 不能用中括号来访问元素, 如果想遍历 set 的话必须用 iterator:

```
set<int> s;
```

```
.....
```

```
for (set<int>::iterator it = s.begin();  
     it != s.end(); it++) {  
    cout << *it << endl;  
}
```

# 映射 (map)

```
#include <map>
```

```
map<key, value> mp;
```



---

## 关于 map

- ▶ map 是照特定顺序存储由 key 和 value 的组合形成的元素的容器
- ▶ map 中元素按照 key 进行排序
- ▶ map 中的每个 key 都是唯一的
- ▶ 另外还有支持多个元素具有等效键的 multimap

---

## map 使用方法 (1)

- ▶ `mp.size()`: 获取 mp 中元素个数  $O(1)$
- ▶ `mp.empty()`: 判断 mp 是否为空  $O(1)$
- ▶ `mp.clear()`: 清空 mp  $O(1)$
- ▶ `mp.find(key)`: 在 mp 中查找一个 key 并返回其 iterator, 找不到的话返回 `s.end()`  $O(\log n)$
- ▶ `mp.count(key)`: 在 mp 中查找 key 的数量, 因为 map 中 key 唯一, 所以只会返回 0 或 1  $O(\log n)$

---

## map 使用方法 (2)

- ▶ `xxx = mp[key]`: 返回 `mp` 中 `key` 对应的 `value`。如果 `key` 不存在, 则返回 `value` 类型的默认构造器 (default constructor) 所构造的值, 并该键值对插入到 `mp` 中  $O(\log n)$
- ▶ `mp[key] = xxx`: 如果 `mp` 中找不到对应的 `key` 则将键值对 (`key: xxx`) 插入到 `mp` 中, 如果存在 `key` 则将这个 `key` 对应的值改变为 `xxx`  $O(\log n)$

## map 使用方法 (3)

- ▶ 和 set 一样, map 的遍历也必须通过 iterator 进行, 具体如下:

```
map<int, int>::iterator it;
for (it = mp.begin(); it != mp.end(); it++) {
    cout << it->first << " "
         << it->second << endl;
}
```

- ▶ 上面的 `it->first` 代表的是 key,
- ▶ `it->second` 代表的是 value

# 基础算法 (algorithm)

```
#include <algorithm>
```

`sort(first, last, compare)`

- ▶ `first`: 排序起始位置 (指针或 iterator)
- ▶ `last`: 排序终止位置 (指针或 iterator)
- ▶ `compare`: 比较方式
- ▶ `sort` 排序的范围是 `[first, second)`, 时间为  $O(n \log n)$
- ▶ 调用 `sort` 时前两个参数必须有, 最后一个参数可有可无, 没有时按升序排序

`sort(first, last, compare)`

- ▶ 数组的排序方法如下:

```
int a[4] = {1, 3, 2, 4};  
sort(a, a + 4);
```

排序结果:

1

- ▶ vector 的排序方法如下:

```
vector<int> v;  
for (int i = 3; i >= 0; i--)  
    v.push_back(i);  
sort(v.begin(), v.end());
```

2

3

4

### sort 中 compare 参数的使用方法

- 在默认排序方法无法满足需要时，我们需要手写一个比较函数作为 sort 的第三个参数：

```
bool comp(const int &a, const int &b) {  
    return a > b;  
}
```

```
int main() {  
    int a[4] = {1, 3, 2, 4};  
    sort(a, a + 4, comp);  
  
    .....
```

```
    return 0;  
}
```

排序结果:

4

3

2

1



`lower_bound(first, last, value)`

- ▶ `first`: 查找起始位置 (指针或 iterator)
- ▶ `last`: 查找终止位置 (指针或 iterator)
- ▶ `value`: 查找的值
- ▶ `lower_bound` 查找的范围是 `[first, second)`, 返回的是序列中第一个不比 `value` 小的元素的位置, 时间为  $O(\log n)$
- ▶ 如果序列内所有元素都比 `value` 小, 则返回 `last`

`lower_bound(first, last, value)`

- ▶ 数组的查找方法如下:

```
int *pos = upper_bound(a, a + 4, 2);
```

查找结果:

- ▶ vector 的查找方法如下:

```
vector<int>::iterator it =  
lower_bound(v.begin(), v.end(), 2);
```

\*pos == 2

\*it == 2

`upper_bound(first, last, value)`

- ▶ `first`: 查找起始位置 (指针或 iterator)
- ▶ `last`: 查找终止位置 (指针或 iterator)
- ▶ `value`: 查找的值
- ▶ `upper_bound` 与 `lower_bound` 相似, 唯一不同的地方在于 `upper_bound` 查找的是序列中第一个大于 `value` 的元素
- ▶ 需要注意的地方时 `lower` 返回的结果可能等于 `value`, 但 `upper` 返回的结果一定大于 `value`

### `next_permutation(first, second)`

- ▶ `first`: 排序起始位置 (指针或 iterator)
- ▶ `last`: 排序终止位置 (指针或 iterator)
- ▶ `next_permutation` 将 `[first, last)` 范围内的元素重新排列为下一个按字典顺序排列的更大排列
- ▶ 如果可以找到下一个排序则返回 `true`, 如果找不到了的话返回 `false` 并将序列调整为字典序最小的序列
- ▶ 时间复杂度为  $O((\text{second} - \text{first}) / 2)$

`next_permutation(first, second)`

► 使用方法如下：

```
int a[3] = {0, 1, 2};
do {
    for (int i = 0; i < 3; i++) {
        printf("%d%c", a[i],
               i == 2 ? '\n' : ' ');
    }
} while (next_permutation(a, a + 3));
```

output:

0 1 2

0 2 1

1 0 2

1 2 0

2 0 1

2 1 0

► vector 使用方法略

`prev_permutation(first, second)`

- ▶ `first`: 排序起始位置 (指针或 iterator)
- ▶ `second`: 排序终止位置 (指针或 iterator)
- ▶ `prev_permutation` 与 `next_permutation` 类似, 唯一的不同时 `prev` 每次将范围内的元素重新排列为下一个按字典顺序排列的更小排列

---

## 其他的一些函数

- ▶ `max(val1, val2)`: 返回更大的数
- ▶ `min(val1, val2)`: 返回更小的数
- ▶ `unique(first, last)`: 移除 `[first, last)` 内连续重复项
- ▶ `stable_sort(first, last, compare)`: 保留具有等效值的元素的相对顺序的排序
- ▶ ○ ○ ○ ○ ○ ○

## 常用的 STL 头文件

- ▶ `#include <string>`
- ▶ `#include <vector>`
- ▶ `#include <queue>`
- ▶ `#include <stack>`
- ▶ `#include <set>`
- ▶ `#include <map>`
- ▶ `#include <algorithm>`





洛谷 P2580 于是他错误的点名开始了 (1000ms / 128MB)

输入：

第一行一个整数  $n$ ，表示班上人数。接下来  $n$  行，每行一个字符串表示其名字（互不相同，且只含小写字母，长度不超过 50）。第  $n+2$  行一个整数  $m$ ，表示教练报的名字的数量。接下来  $m$  行，每行一个字符串表示教练报的名字（只含小写字母，且长度不超过 50）。 $n \leq 10000$ ， $m \leq 100000$

输出：

对于每个教练报的名字，输出一行。如果该名字正确且是第一次出现，输出“OK”，如果该名字错误，输出“WRONG”，如果该名字正确但不是第一次出现，输出“REPEAT”。（均不加引号）

洛谷 P2580 于是他错误的点名开始了 (1000ms / 128MB)

样例输入：

5  
a  
b  
c  
ad  
acd  
3  
a  
a  
e

样例输出：

OK  
REPEAT  
WRONG

```
int num(0), root = 1;
for (re int i = 0; str[i]; ++i) {
    num = str[i] - 'a';
    if(!a[root].next[num]) a[root].next[num] = ++top;
    root = a[root].next[num];
}
a[root].exist = true;
return ;
}

inline int Trie_search() { //普通search操作, 查询是否存在该字符串
    int num(0), root = 1;
    for (re int i = 0; str[i]; ++i) {
        num = str[i] - 'a';
        if(!a[root].next[num]) return 0; //若无法查询到后继的字母, 直接退出
        root = a[root].next[num];
    }
    if(!a[root].exist) return 0; //若此处不存在字符串, 返回“WRONG”
    else if(a[root].count) return 2; //若此处字符串已被查询, 返回“REPEAT”
    a[root].count = true; //若以上均不满足, 说明合法
    return 1; //此字符串标记为已查询, 返回“OK”
}

int main() {
    //freopen("Trie_Tree.in", "r", stdin);本地测试用
    getNum(n);
    for (re int i = 1; i <= n; ++i) { //插入
        memset(str, 0, sizeof str);
        getString(str);
        Trie_insert();
    }
    getNum(m):
```

---

## STL 解法

- ▶ 声明一个 key 为 string, value 为 int 的 map。
- ▶ key 代表同学的名字
- ▶ value 代表被教练点到的次数
- ▶ 设定初始情况下每一个名字对应的次数都为 1

---

## STL 解法

- ▶ 对于每一次点名，在 map 中查找对应的 key
- ▶ 如果 key 对应的 value 为 0 则输出「WRONG」
- ▶ 如果对应的 value 为 1 则输出「OK」并将 value + 1
- ▶ 如果对应的 value 为 2 则输出「REPEAT」

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

```
int main() {
    int n;
    map<string, int> mp;
    string name;

    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> name;
        mp[name] = 1;
    }
}
```

```
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> name;
        int flag = mp[name];
        if (flag == 0) {
            cout << "WRONG" << endl;
        } else if (flag == 1) {
            cout << "OK" << endl;
            mp[name]++;
        } else {
            cout << "REPEAT" << endl;
        }
    }

    return 0;
}
```

## 关于程序运行时间

- ▶ 一般情况下用 `cin/cout` 进行数据读写要比 `scanf/printf` 慢得多，所以在输入输出文件较大的情况下用 `cin/cout` 可能会超时
- ▶ `cin/cout` 效率低的原因之一是它要与 `stdio` 保持同步，以免在二者混用的时候出现问题
- ▶ 因此我们可以通过关闭与 `stdio` 的同步来提高 `cin/cout` 的速度，不过这时我们不能再两者混用了，不然会出错
- ▶ 关闭同步代码：`ios::sync_with_stdio(false);`



### 关于程序运行时间

- ▶ `map/set` 因为要维持内部元素排列有序所以在时间上会多一些开销
- ▶ 如果我们不需要维持排列有序的话可以使用 C++ 11 提供的 `unordered_map/unordered_set`
- ▶ 这两种容器分别在 `#include <unordered_map>` 和 `#include <unordered_set>` 里

### 关于程序运行时间

- ▶ `unordered_map` 和 `unordered_set` 的查询操作的平均时间复杂度为  $O(1)$ ，最坏时间复杂度为  $O(n)$ ，可以大大提高查询效率
- ▶ 不过如果比赛不提供 C++ 11 的环境的话这两种容器就无法使用了

### 关于程序运行时间

- ▶ 对 `vector` 进行大量 `push_back` 操作会产生很大的时间开销，因为每次 `push_back` 都需要请求新内存空间，并且在当前 `vector` 长度大于 `capacity`（容量）值时会对整个 `vector` 重新分配内存
- ▶ 因此在使用 `vector` 之前可以通过 `.reserve(n)` 将 `vector` 的 `capacity` 设定为一个足够大的数 `n`，这样可以大大提高 `push_back` 的效率
- ▶ `reserve` 不会影响 `vector` 中的元素

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

```
int main() {
    int n;
    unordered_map<string, int> mp;
    string name;
```

```
    ios::sync_with_stdio(false);
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> name;
        mp[name] = 1;
    }
```

优化前

评测状态

Accepted 100

用时: 1640ms / 内存: 18386KB

优化后

评测状态

Accepted 100

用时: 868ms / 内存: 15175KB

---

**THANKS FOR LISTENING**

**\_liet**