C++ 算法：哈希表

一、 引言

1.1 问题引入：字符串的查重

【例17-3 字符串哈希】给定 N 个字符串，求共有多少个**不同**的字符串。

常规思路：每拿到一个新字符串，都去已有的列表中从头到尾比对一遍。如果字符串数量达到十万、百万级别，这种时间复杂度为 $O(N^2)$ 的方法必将超时。

我们需要**看一眼字符串的名字，就能瞬间知道它应该放在哪里，以及它是否已经存在。**

1.2 核心思想：图书管理员

想象你是一个图书管理员，面前有海量的书（字符串）。你不想每次都跑遍整个图书馆找书。你希望有一个**规则**，让你一看到书名，就能立刻算出它应该放在编号为几的书架上。

- **字符串** -> *(通过某个神奇规则)* -> **唯一的书架编号**

这个神奇的规则，就是**哈希函数**。通过它，我们可以将查找的时间复杂度从线性的 $O(N)$ 降低到近乎 $O(1)$ 。

二、 核心原理解析

一个完整的哈希表系统，由两个核心部分组成：**哈希函数** 和 **桶数组（加冲突解决机制）**。

2.1 哈希函数：将字符串变成数字

我们要解决的第一个问题是：如何把一个字符串（如 `"abc"`）转换成一个唯一的数字索引（键值/哈希码）？

- **错误思路**：直接将字符的 ASCII 码相加。这样 `"abc"` 和 `"cba"` 的结果一样，会导致严重的混淆。
- **正确思路：字符串哈希 (进制转换法)**。位置很重要！就像十进制的 `123`（$1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$）。我们把字符串看作一个 **P进制数**。

哈希公式：$hash("abc") = ('a' * P^2 + 'b' * P^1 + 'c' * P^0)$

- P 通常取质数（如 131），MOD 取一个很大的质数（如 $10^9 + 7$）以防止溢出并压缩数值范围。

代码实现（迭代法）：
我们不需要真去算 P 的次方，只需像从高位读数一样，每次将旧结果乘以 P，再加上新字符即可。

```
const int P = 131;
const long long MOD = 1e9 + 7;

int hash_string(const string& str) {
    long long h = 0;
    for (char c : str) {
        h = (h * P + c) % MOD; // 迭代公式
    }
    return h;
}
```

2.2 桶数组与哈希冲突

算出哈希值后，我们需要把它放入实际的存储空间——**桶数组**。
为了把巨大的哈希值塞进有限的数组里，我们必须进行取模运算：`index = hash_val % SIZE`（`SIZE` 为桶数组的大小）。

哈希冲突：
根据鸽巢原理，必定会有两个**不同**的字符串，计算出的 `index` **完全相同**。这就好比图书管理员发现，两本不同的书被分配到了同一个书架号。难道我们直接扔掉一本吗？

2.3 解决冲突：链地址法

遇到冲突时，最经典的解决方案是**链地址法**。
书架不能只放一本书了。我们在每个书架的位置放一个“箱子”或“链表”。所有被分配到这个书架号的书，统统扔进这个箱子里。

数据结构设计：

```
// SIZE 是桶的数量
// vector<string> 就是那个"箱子"，里面可以装多本哈希值冲突的书
vector<vector<string>> hashTable(SIZE);
```

查找/插入流程：

1. 计算新字符串 `s` 的下标：`index = hash_string(s) % SIZE`。
2. 直接走到第 `index` 个箱子前。
3. **遍历这个箱子** `hashTable[index]`。
    - 如果找到了 `s`，说明重复了，什么都不做。
    - 如果找遍了箱子也没找到，说明是个全新的字符串，把它装进去：`hashTable[index].push_back(s)`。

只要哈希函数设计得好，冲突就会很少，每个箱子里的书极少，查找速度依然是 $O(1)$。但如果哈希函数很糟，所有书都挤在一个箱子里，哈希表就会退化成一个长数组，查找速度降回 $O(N)$。

三、 STL: `unordered_map`

手写哈希表能深刻理解原理，但在实战中，C++ 标准库（STL）已经为我们准备好了封装的哈希表——`unordered_map`。

它隐藏了复杂的哈希函数计算、内存分配和冲突处理，提供了一套极简、高效的操作接口。

3.1 定义与核心特性

- **头文件**：`#include <unordered_map>`
- **特性**：键（Key）到值（Value）的映射，键是**唯一**的。内部元素是**无序**的。
- **定义语法**：`unordered_map<KeyType, ValueType> umap;`
    - 例如：`unordered_map<string, int> count;`（通常用于统计字符串出现的次数）

3.2 操作符：`[]`

当我们写下 `umap[key]` 时，会发生：

1. **如果 `key` 已经存在**：
   它返回该 `key` 对应的值（Value）的引用。你可以直接读取或修改它。
   `cout << umap["apple"];` 对应输出 apple 的数量
   `umap["apple"]++;` 对应 apple 的数量加 1
2. **如果 `key` 不存在（重要）**：
   它会**自动**在哈希表中创建一个新的键值对 `(key, 默认值)`！如果是 `int`，默认值就是 0。
   `umap["banana"]++;` // 如果原本没有 banana，它会创建并设为 `int` 类型的初始值 0，然后执行++变成1。

3.3 其他常用成员函数

- **查找**：`umap.count(key)`。存在返回1，不存在返回0。比 `[]` 更安全，因为 `count` **不会**自动创建不存在的键。
- **插入**：`umap.insert({"apple", 10})`。如果键已存在，插入会失败。
- **删除**：`umap.erase(key)`。删除指定键的元素。
- **状态**：`umap.size()`（元素个数），`umap.empty()`（是否为空）。

3.4 遍历

使用基于范围的 `for` 循环遍历，每次拿到的是一个 `pair`（包含 `.first` 和 `.second`）。

```
unordered_map<string, int> fruit = {{"apple", 5}, {"banana", 3}};

for (const auto& pair : fruit) {
    cout << "水果: " << pair.first << ", 数量: " << pair.second << endl;
}
```

四、 案例：Cities and States (B0250)

题意简述：寻找特殊的城市对。城市1的前两个字母 == 城市2的州代码，且 城市2的前两个字母 == 城市1的州代码，且两者不在同一个州。

```
int main() {
    int n;
    cin >> n;

    // 键：组合字符串， 值：该组合出现的次数
    unordered_map<string, int> counts;
    long long ans = 0;

    for (int i = 0; i < n; i++) {
        string city, state;
        cin >> city >> state;

        string city_prefix = city.substr(0, 2); // 截取前两个字母
```

```
        if (city_prefix != state) { // 必须来自不同的州
            string self_key = city_prefix + state;   // 我自己的特征
            string partner_key = state + city_prefix; // 我需要的伴侣特征

            // 查找伴侣：去哈希表里找，我需要的伴侣之前出现了几次？
            // 注意利用了 [] 运算符的自动创建特性，如果不存在就返回0，不会报错
            ans += counts[partner_key];

            // 登记自己：把自己加入哈希表，供后面的城市查找
            counts[self_key]++;
        }
    }

    cout << ans << endl;
    return 0;
}
```