**程序设计与算法综合训练**

**实验报告**

|  |  |
| --- | --- |
| **实验名称**： | 通讯录查询系统的设计与实现 |
| **专业**： | 机器人工程 |
| **班级**： | 机器人工程三班 |
| **学号**： | WA2224013 |
| **姓名**： | 郭义月 |

**目录**

[一、 实验内容及要求 4](#_Toc184205460)

[1.1 实验目的 4](#_Toc184205461)

[1.2 实验内容 4](#_Toc184205462)

[1.3 实验要求 4](#_Toc184205463)

[二、 哈希表相关知识 5](#_Toc184205464)

[2.1 哈希表基本概念 5](#_Toc184205465)

[2.2 哈希函数 5](#_Toc184205466)

[2.3 哈希冲突 5](#_Toc184205467)

[2.4 冲突解决方法 5](#_Toc184205468)

[2.5 负载因子与哈希表大小 6](#_Toc184205469)

[2.6 哈希表的基本操作 6](#_Toc184205470)

[三、 定义相关的类与结构体 6](#_Toc184205471)

[3.1 Employee结构体 6](#_Toc184205472)

[3.2 HashTable 类 7](#_Toc184205473)

[3.2.1 成员变量 7](#_Toc184205474)

[3.2.2 hashFunction 函数 7](#_Toc184205475)

[3.2.3 quadraticProbing 函数 7](#_Toc184205476)

[3.2.4 createContact 函数 8](#_Toc184205477)

[3.2.5 displayContacts 函数 8](#_Toc184205478)

[3.2.6 deleteContact 函数 8](#_Toc184205479)

[3.2.7 searchContact 函数 9](#_Toc184205480)

[3.2.8 editContact 函数 9](#_Toc184205481)

[3.2.9 clearContacts 函数 10](#_Toc184205482)

[3.3 主函数 10](#_Toc184205483)

[四、 实验结果 10](#_Toc184205484)

[4.1 新建联系人 11](#_Toc184205485)

[4.2 显示联系人 11](#_Toc184205486)

[4.3 查找联系人 11](#_Toc184205487)

[4.4 删除联系人 12](#_Toc184205488)

[4.5 修改联系人 13](#_Toc184205489)

[4.6 清空联系人 13](#_Toc184205490)

[4.7 退出程序 13](#_Toc184205491)

[五、 实验总结与常见问题汇总 14](#_Toc184205492)

[5.1 实验总结 14](#_Toc184205493)

[5.2 常见问题汇总 14](#_Toc184205494)

# 实验内容及要求

## 实验目的

本实验的目的是通过设计并实现一个员工通讯录管理系统。理解哈希表的基本原理及其操作，包括插入、删除和查找。通过实现二次探测再散列法来解决哈希冲突，进一步掌握冲突解决策略。同时，通过编写程序实现根据电话号码查找员工信息，并将通讯录数据保存到文件中，确保数据持久化。

## 实验内容

为某个单位建立一个员工通讯录管理系统，可以方便查询每一个员工的电话与地址。利用哈希表（散列表）存储，设计并实现通讯录查找系统。

## 实验要求

(1)每个记录有下列数据项：电话号码、用户名、地址；

(2)从键盘输入各记录，分别以电话号码为关键字建立散列表；

(3)采用二次探测再散列法解决冲突（两个不同的关键字，由于散列函 数值相同，因而被映射到同一表位置上为冲突。）；

(4)查找并显示给定电话号码的记录；

(5)通讯录信息文件保存。

# 哈希表相关知识

在处理员工通讯录管理系统时，哈希表是一个非常合适的数据结构，特别是对于高效的查找、插入和删除操作。

## 哈希表基本概念

哈希表是一种基于数组的数据结构，用于将数据映射到固定大小的数组中。哈希表通过哈希函数将数据的键（如电话号码）映射到数组的索引位置，从而实现高效的查找、插入和删除操作。

## 哈希函数

哈希函数是哈希表的核心，它将一个输入（例如电话号码）转换成一个索引。理想情况下，哈希函数应当使得数据均匀分布在哈希表中，从而避免发生过多的冲突。一个好的哈希函数能够提高哈希表的性能，确保操作的时间复杂度接近O(1)。

## 哈希冲突

哈希冲突是指不同的键经过哈希函数计算后，得到相同的索引位置。当两个或多个数据项被映射到哈希表中的同一个位置时，就发生了哈希冲突。冲突会导致性能下降，因此需要采取措施来解决。

## 冲突解决方法

常见的冲突解决方法有两种：

**开放地址法**：在哈希表中，遇到冲突时，查找下一个空的位置来存储数据。常用的开放地址法包括线性探测、二次探测和双重哈希。

**链表法**：每个数组位置存储一个链表，所有哈希到同一位置的元素通过链表存储，这样就可以避免冲突。

在本实验中，我们采用了 **二次探测再散列法** 来解决冲突。具体做法是：当发生冲突时，不是按固定的间隔（如线性探测）查找下一个空位，而是通过一个二次函数来调整步长，即寻找的下一个位置由公式 (index + i^2) % table\_size 确定，其中 i 是探测次数。这种方法可以减少相邻冲突，避免性能下降。

## 负载因子与哈希表大小

负载因子（Load Factor）是哈希表中元素个数与表大小的比值。负载因子过高会导致大量冲突，降低查找效率，因此需要动态调整哈希表的大小。通常，当负载因子超过某个阈值时（如0.7），需要扩展哈希表的大小，并重新计算所有元素的位置。

## 哈希表的基本操作

**插入**：通过哈希函数计算索引，将数据插入该位置。如果发生冲突，则使用二次探测法查找下一个空位并插入。

**查找**：通过哈希函数计算索引，直接定位到该位置。如果当前位置的元素不是目标元素，则根据二次探测法继续查找。

**删除**：与查找类似，找到目标元素后将其删除，并通过二次探测法保证删除后的位置能够正确找到。

通过这些哈希表的知识，你可以有效地实现一个通讯录管理系统，能够快速查找和存储员工的电话号码、姓名和地址。

# 定义相关的类与结构体

## Employee结构体

根据实验的要求，通讯录里面每个人都要有姓名，电话和住址，所以定义一个结构体存储员工的基本信息，其中isOccupied是一个布尔值，标志这个 Employee 是否已被占用（即是否存储了有效的员工信息）。初始值为 false，表示该位置尚未存储员工信息。

struct Employee {

string name;

string phone;

string address;

bool isOccupied = false;

};

## HashTable 类

这个类定义一个哈希表来实现实验中要求的插入，删除以及查询的操作

### 成员变量

private:

vector<Employee> table;

int size;

table：是一个 vector，用于存储哈希表的所有 Employee 对象。通过这种方式来模拟哈希表的“桶”。

size：哈希表的大小，决定了存储数据的数量上限。

### hashFunction 函数

int hashFunction(const string &phone) {

int hash = 0;

for (char c : phone) hash = (hash \* 31 + c) % size;

return hash;

}

这是哈希函数，负责将电话号码转换为哈希表的索引。具体实现方法是：

对电话号码中的每个字符进行遍历，并根据字符的 ASCII 值计算一个哈希值。31 是一个常用的哈希乘数（有助于生成均匀分布的哈希值）。 size 确保生成的哈希值始终落在哈希表的有效范围内。

### quadraticProbing 函数

int quadraticProbing(int index, int attempt) {

return (index + attempt \* attempt) % size;

}

这是解决哈希冲突的二次探测法。哈希冲突发生时，使用该方法根据探测次数（attempt）调整索引位置。二次探测使得冲突位置的分布更加分散，减少冲突的聚集。

### createContact 函数

void createContact(const string &phone, const string &name, const string &address) {

int index = hashFunction(phone);

int attempt = 0;

while (table[index].isOccupied) {

index = quadraticProbing(index, ++attempt);

}

table[index] = {name, phone, address, true};

cout << "Contact created successfully!\n";

}

用于创建新的联系人。首先通过电话号码计算哈希值并定位到一个索引。如果该位置已经被占用（即 isOccupied 为 true），则通过二次探测继续寻找下一个空位置。找到空位置后，将新的员工信息插入到该位置。

### displayContacts 函数

void displayContacts() {

for (const auto &record : table) {

if (record.isOccupied) {

cout << "Name: " << record.name

<< ", Phone: " << record.phone

<< ", Address: " << record.address << "\n";

}

}

}

该函数遍历哈希表，显示所有已占用位置的联系人信息。如果 isOccupied 为 true，则说明该位置存储了有效的员工信息，接着输出姓名、电话号码和地址。

### deleteContact 函数

void deleteContact(const string &phone) {

int index = hashFunction(phone);

int attempt = 0;

while (table[index].isOccupied && table[index].phone != phone) {

index = quadraticProbing(index, ++attempt);

if (attempt > size) {

cout << "Contact not found.\n";

return;

}

}

if (table[index].isOccupied && table[index].phone == phone) {

table[index] = Employee(); // Reset the entry

cout << "Contact deleted successfully!\n";

} else {

cout << "Contact not found.\n";

}

}

该函数根据电话号码查找并删除联系人：

使用哈希函数定位到对应的索引。如果该位置存储的不是目标联系人，则使用二次探测查找其他可能的位置。一旦找到目标联系人，将其从哈希表中删除，通过将 Employee 结构体重置来清空该位置。

### searchContact 函数

void searchContact(const string &phone) {

int index = hashFunction(phone);

int attempt = 0;

while (table[index].isOccupied && table[index].phone != phone) {

index = quadraticProbing(index, ++attempt);

if (attempt > size) {

cout << "Contact not found.\n";

return;

}

}

if (table[index].isOccupied && table[index].phone == phone) {

cout << "Name: " << table[index].name

<< ", Phone: " << table[index].phone

<< ", Address: " << table[index].address << "\n";

} else {

cout << "Contact not found.\n";

}

}

该函数根据电话号码查找并显示联系人信息：使用哈希函数计算索引并通过二次探测查找目标联系人。如果找到该联系人，显示其姓名、电话和地址，否则显示未找到信息。

### editContact 函数

void editContact(const string &phone) {

int index = hashFunction(phone);

int attempt = 0;

while (table[index].isOccupied && table[index].phone != phone) {

index = quadraticProbing(index, ++attempt);

if (attempt > size) {

cout << "Contact not found.\n";

return;

}

}

if (table[index].isOccupied && table[index].phone == phone) {

string newName, newAddress;

cout << "Enter new name: ";

getline(cin, newName);

cout << "Enter new address: ";

getline(cin, newAddress);

table[index].name = newName;

table[index].address = newAddress;

cout << "Contact updated successfully!\n";

} else {

cout << "Contact not found.\n";

}

}

该函数允许修改已有联系人的信息：根据电话号码查找联系人，找到后允许用户输入新的姓名和地址并更新原始数据。

### clearContacts 函数

void clearContacts() {

table.clear();

table.resize(size);

cout << "All contacts cleared successfully!\n";

}

该函数用于清空哈希表中的所有联系人记录。

## 主函数

在 main 函数中，初始化一个大小为 10 的哈希表，并显示菜单，可以选择进行联系人管理操作。通过 cin 和 getline 获取用户输入。

# 实验结果

分别在每个函数的执行前后打上断点，方便调试与观察

## 新建联系人

打断点可以看出，目前输入的员工的手机号与名字和住址，逐步调试后发现可以顺利实现新建联系人的功能

|  |  |
| --- | --- |
|  |  |

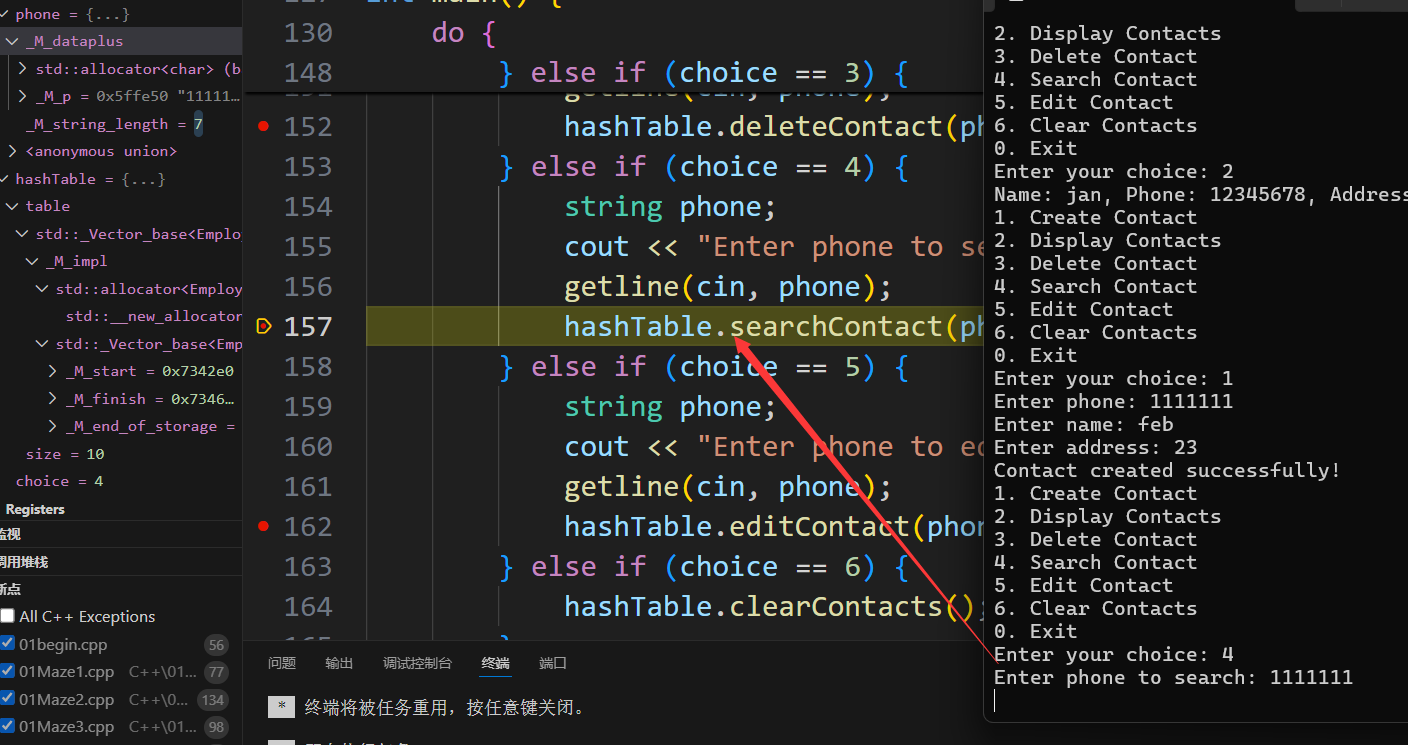
## 显示联系人

可以按照代码中的设置顺利显示

|  |  |
| --- | --- |
|  |  |

## 查找联系人

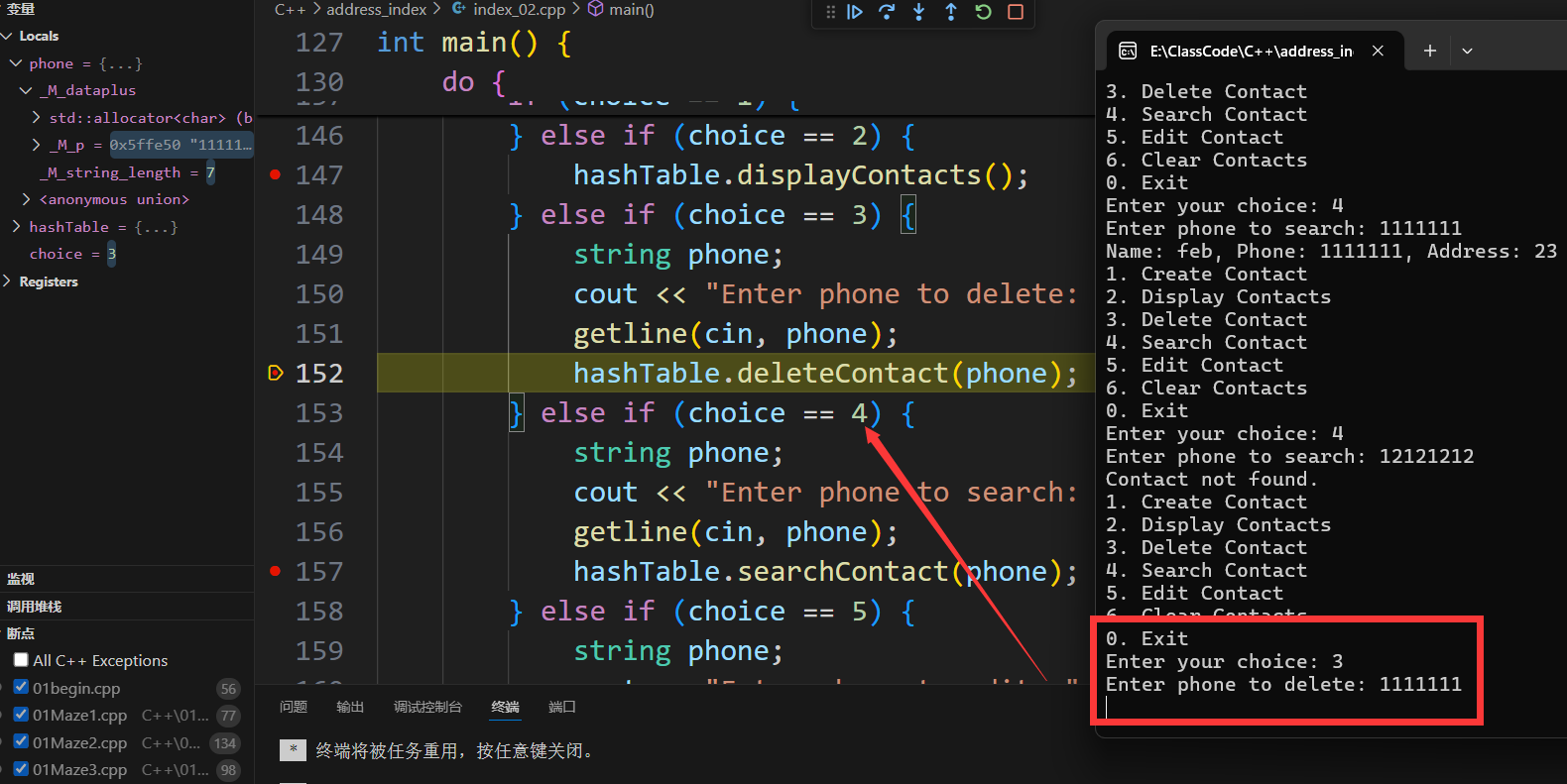
选择4，逐步调试



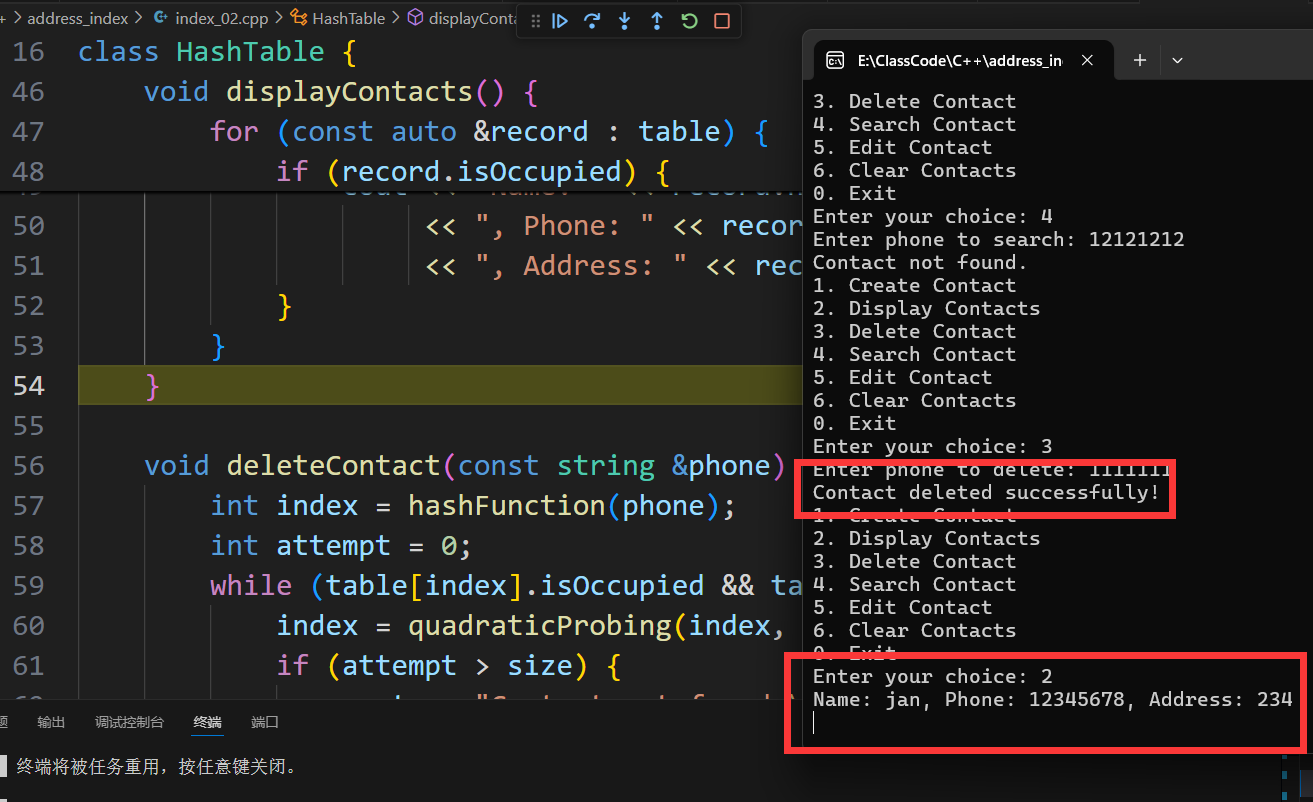
通过电话号码查找联系人，如果联系人存在，就显示其属性，如果不存在，显示无法找到该联系人

|  |  |
| --- | --- |
|  |  |

## 删除联系人



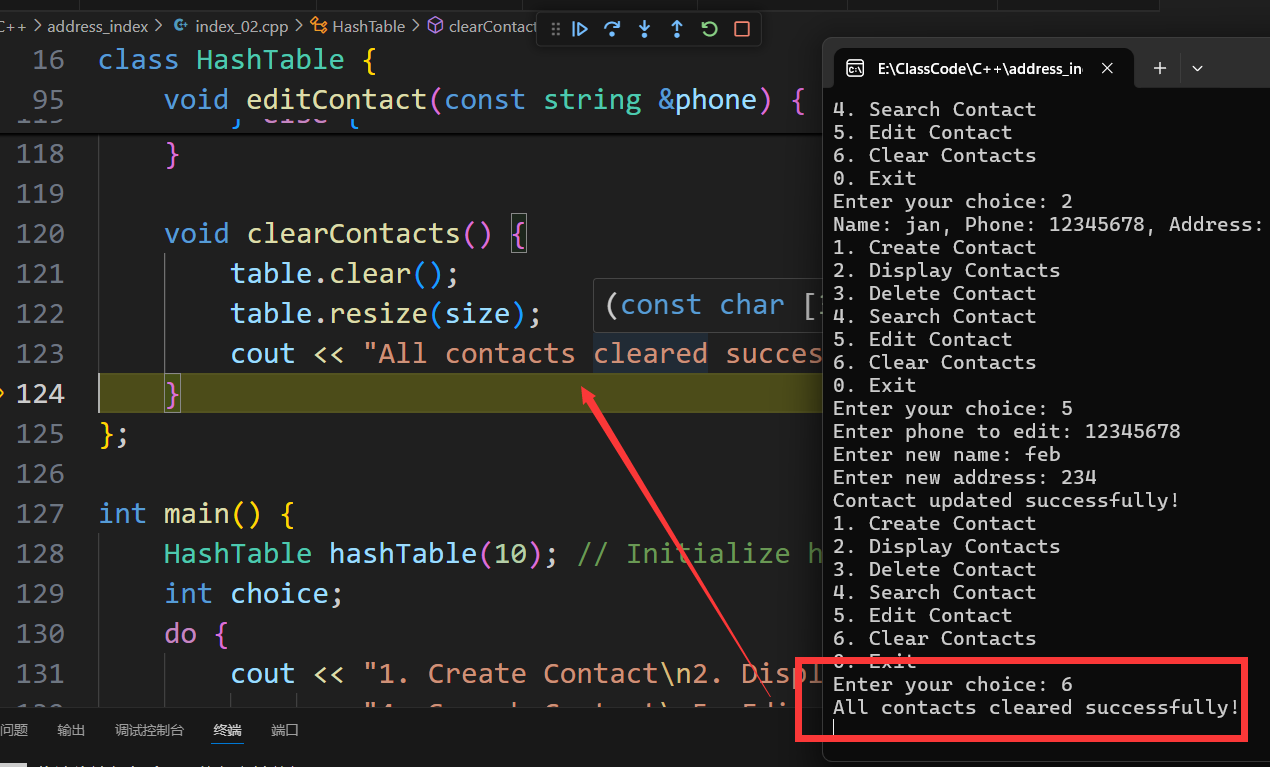
执行后再选择显示联系人之后，之前选择要删除的联系人已经没有了，正确实现删除联系人的功能。



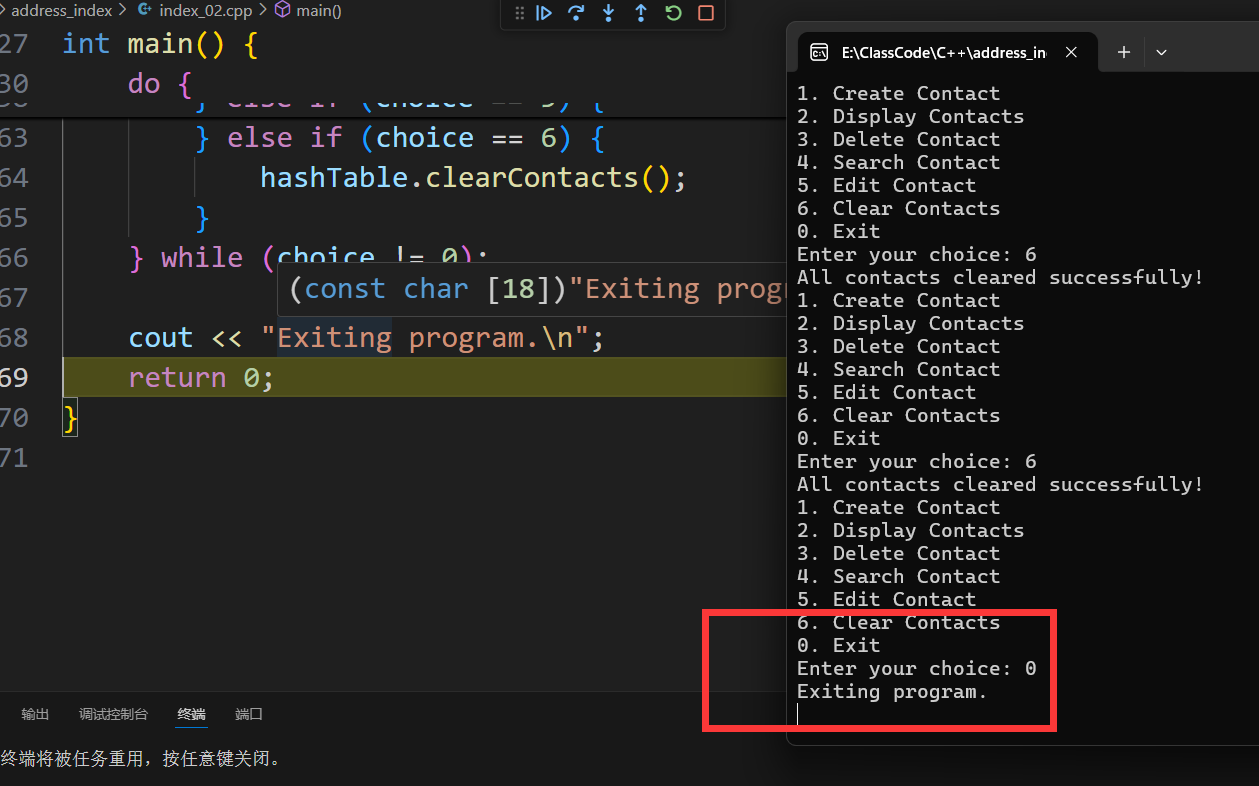
## 修改联系人

|  |  |
| --- | --- |
|  |  |

## 清空联系人



## 退出程序



# 实验总结与常见问题汇总

## 实验总结

完成这个哈希表实现的员工通讯录系统后，对哈希表的理解更深了。哈希表的查找、插入和删除操作效率非常高，能在常数时间内完成，这对处理大量数据很有帮助。在实验中，通过哈希函数和二次探测法解决冲突，学会了如何避免冲突并优化性能。

用类来组织代码也让我体会到了面向对象编程的优势。Employee和 HashTable类让代码结构清晰，逻辑也更易于维护。通过封装数据和方法，代码更加模块化，功能也更容易扩展。实际开发中，面向对象设计能提高代码的可读性和可复用性。

这个项目不仅让我提升了编程能力，还加深了对数据结构和算法的理解。特别是在处理输入输出、错误处理和用户交互时，学到了很多细节和技巧。调试过程中遇到的各种问题也让我意识到测试和代码重构的重要性，逐步优化程序性能。

总的来说，这个实验让我更熟练地使用哈希表和面向对象编程，提升了代码能力，也让我对解决实际问题有了更多思考。

## 常见问题汇总

本任务在定义哈希类的时候很容易出问题，一定要先计划好成员变量与每个函数的设计，否则就会牵一发而动全身，最后不得不定义很多全局变量来解决一些函数里面self没有这个属性的问题，要有提前规划的意识，之后才能事半功倍。