# 程序设计与算法综合训练

# 实验报告

# 实验名称： 迷宫问题

# 专业班级：人工智能二班

# 学号：WA2214014

# 姓名：杨跃浙

目录

[一、实验内容及要求 3](#_Toc12501)

[1.1 实验目的 3](#_Toc19501)

[1.2 实验内容 3](#_Toc5975)

[1.3 实验要求 3](#_Toc25843)

[1.4 实验任务 3](#_Toc6849)

[二、栈和队列 4](#_Toc22947)

[2.1 问题描述 4](#_Toc1929)

[2.2 栈和队列 4](#_Toc11163)

[2.2.1 栈 4](#_Toc24909)

[2.2.2 将栈的数据类型改为与问题匹配的Point 9](#_Toc28711)

[2.2.3 链栈 10](#_Toc6262)

[2.2.4 Point类型的链栈 12](#_Toc13030)

[2.2.5 队列 14](#_Toc13230)

[2.2.6 链队列 14](#_Toc25210)

[三、递归和迭代 15](#_Toc16082)

[3.1 理论 15](#_Toc6732)

[3.2 实验 16](#_Toc26046)

[四、迷宫问题 16](#_Toc27765)

[4.1 DFS递归实现 16](#_Toc21784)

[4.2 DFS迭代实现 18](#_Toc15622)

[4.2（补充） Vector库介绍 20](#_Toc5926)

[4.3 DFS输出所以通路 21](#_Toc11447)

[修改部分的核心代码 24](#_Toc13094)

[4.4 BFS实现输出最短通路 24](#_Toc29096)

[4.5 DFS和BFS 27](#_Toc2849)

[4.5.1 DFS和BFS的差异 27](#_Toc8294)

[4.5.2 A\*算法 27](#_Toc22908)

[五、实验总结 29](#_Toc10685)

# 一、实验内容及要求

## 1.1 实验目的

加深对栈和队列数据结构的理解，强化同学们的逻辑思维能力和动手能力，巩固良好的编程习惯，掌握工程软件设计的基本方法，为后续课程的学习打下坚实基础。

## 1.2 实验内容

问题描述：

以一个m\*n的长方阵表示迷宫，0和1分别表示迷宫中的通路和障碍。设计一个程序，对任意设定的迷宫，求出一条从入口到出口的通路，或得出没有通路的结论。

## 1.3 实验要求

基本要求：

（1）首先实现一个以链表作存储结构的栈类型，然后编写一个求解迷宫的非递归程序。求得的通路以三元组（i，j，d）的形式输出。其中：（i，j）指示迷宫中的一个坐标，d表示走到下一坐标的方向。如，对于下列数据的迷宫，输出一条通路为：（1，1，1），（1，2，2），（2，2，2），（3，2，3），（3，1，2），…。

（2）编写递归形式的算法，求得迷宫中所有可能的通路。

（3）以方阵形式输出迷宫及其通路。

## 1.4 实验任务

任务:实验1的要点总结

1.栈与队列的顺序、链式结构的实现与扩展，以及相关代码的调试。

2.递归与迭代的区别与联系，以及相关代码的调试。

3.深搜与广搜的区别、优缺点。

4.尝试编写用广搜的代码

5.提交程序代码基本要求：

（1）首先实现一个以链表作存储结构的栈类型，然后编写一个求解迷宫的非递归程序。求得的通路以三元组（i，j，d）的形式输出。其中：（i，j）指示迷宫中的一个坐标，d表示走到下一坐标的方向。如，对于下列数据的迷宫，输出一条通路为：（1，1，1），（1，2，2），（2，2，2），（3，2，3），（3，1，2），…。

（2）编写递归形式的算法，求得迷宫中所有可能的通路。

（3）以方阵形式输出迷宫及其通路。

# 二、栈和队列

## 2.1 问题描述：

以一个m\*n的长方阵表示迷宫，0和1分别表示迷宫中的通路和障碍。设计一个程序，对任意设定的迷宫，求出一条从入口到出口的通路，或得出没有通路的结论。

深度优先搜索是指从一个结点开始尽可能深入到图的每个分支中，直到找到目标或者无法继续深入为止。

具体过程如下：

1.从图的一个结点开始遍历。

2.若当前结点有未访问过的相邻结点，则选择其中一个结点，继续深入遍历。

3.若当前结点没有未访问过的相邻结点，则返回到上一个结点，并在该结点的其他未访问过的相邻结点中选择一个继续遍历。

4.重复2-3步骤，直到遍历完所有结点或找到目标结点。

深度优先搜索解决此迷宫问题：

无论是递归还是非递归深度优先搜索都运用到了栈：

1.创建一个空栈，并将起点压入栈中。

2.当栈不为空时，判断栈顶元素，如果该元素为终点，则搜索结束，否则将其标记为已访问。

3.检查当前结点的相邻结点是否可行（未被访问且不是障碍物），如果可行则将其压入栈中；若不可行则将该结点出栈。

4.重复步骤2-3，直到找到终点或栈为空。

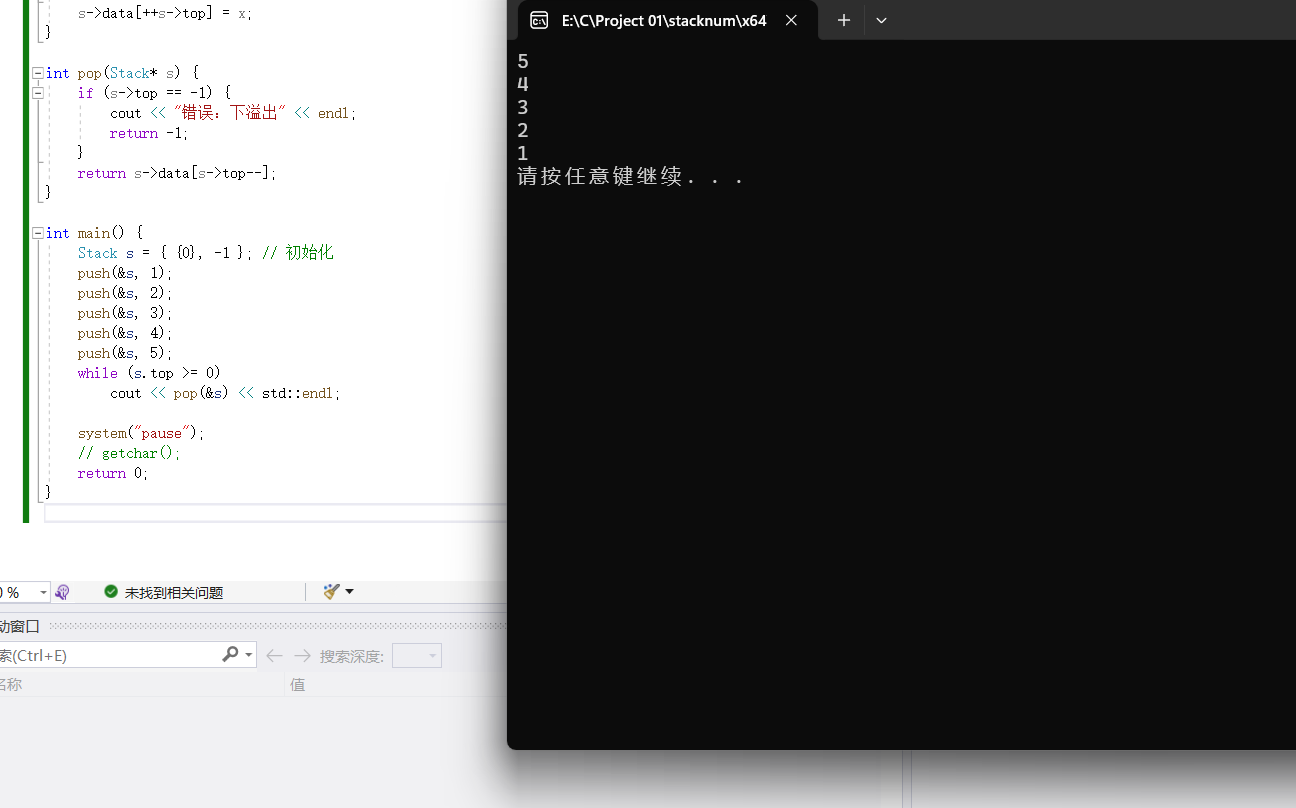
5.如果栈为空且未找到终点，则无解。

## 2.2 栈和队列

### 2.2.1 栈

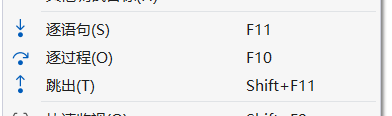
#### 运行和调试：

运行stacknum 观察输出情况

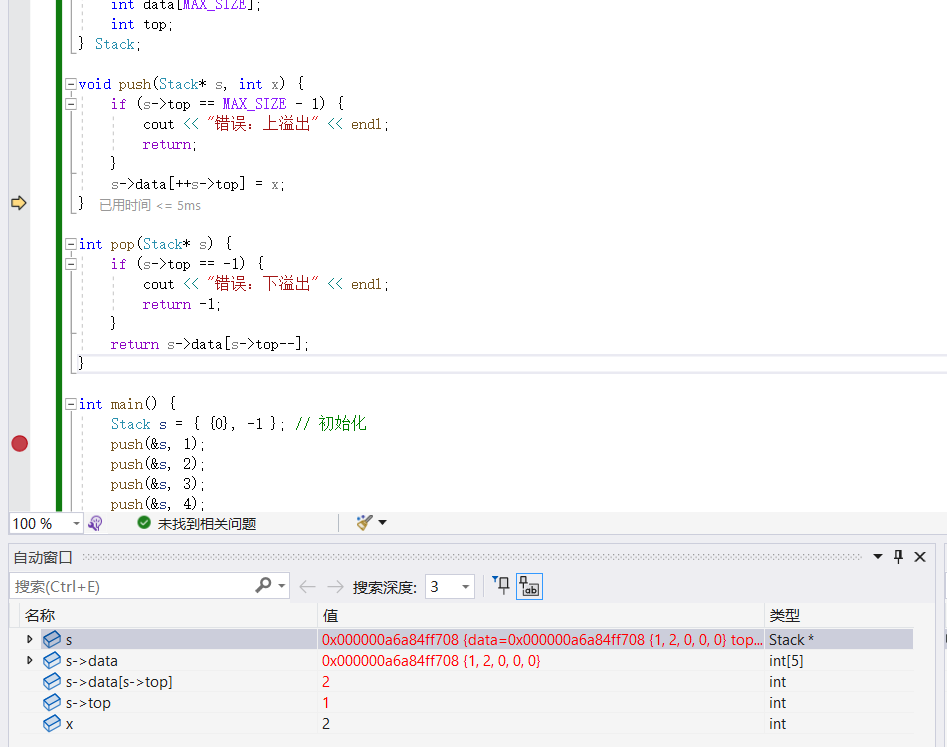


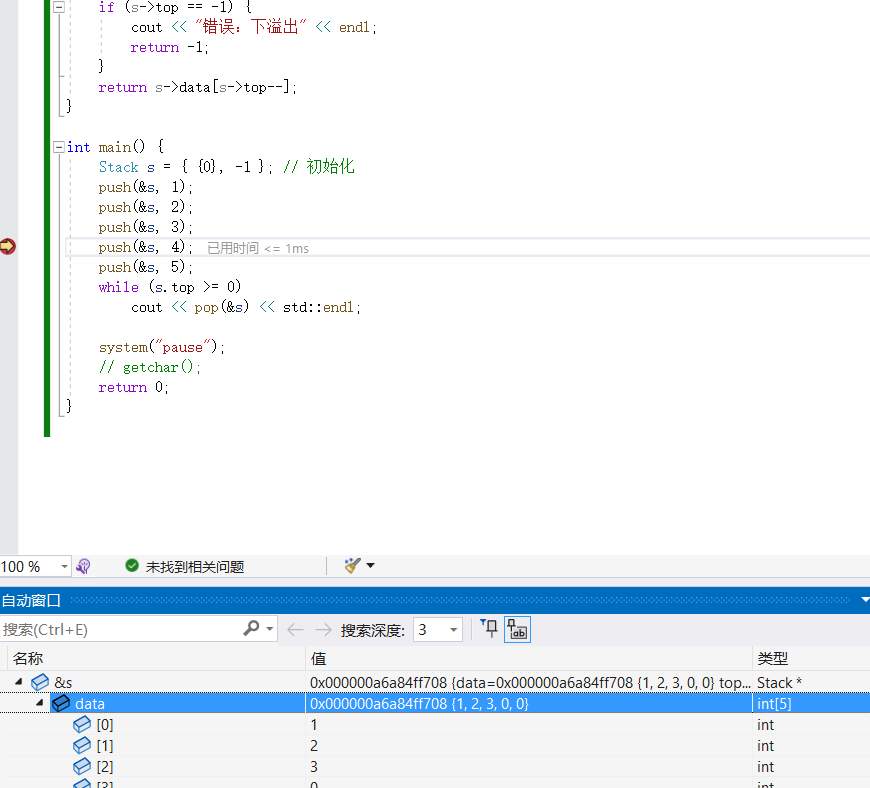
对代码进行debug

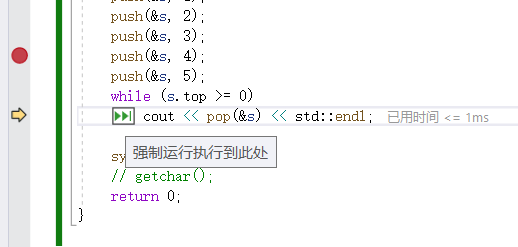
观察进栈出栈情况



版本vs2022中 逐语句调试为F11 逐过程为F10 调试为F5

可以用逐语句调试观察到栈内元素变化

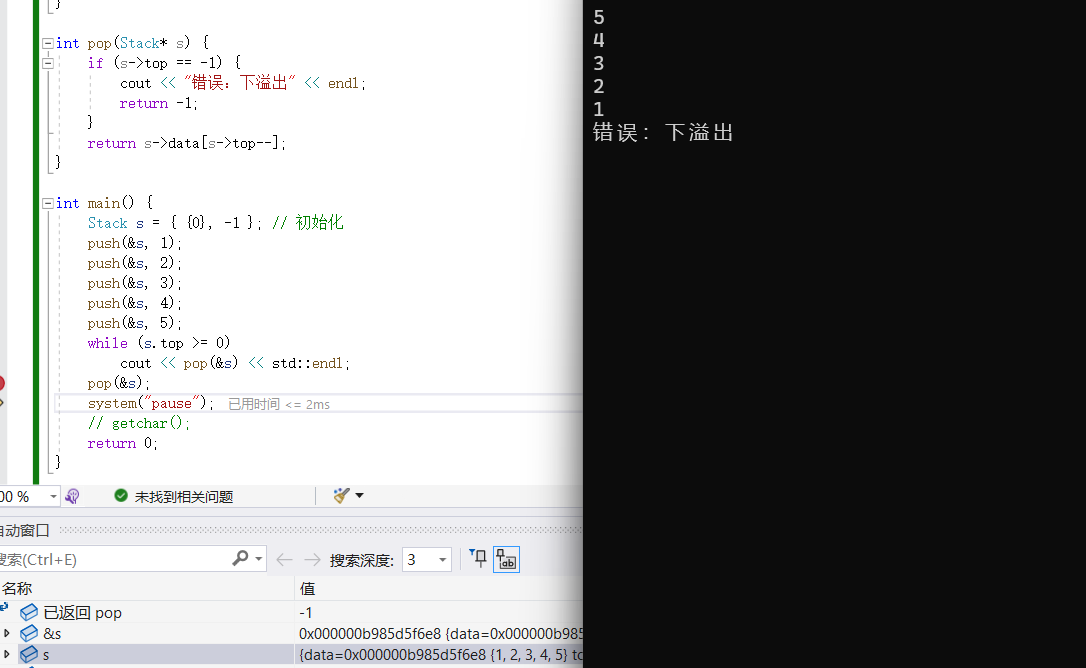
也可以用断点的形式呈现 通过在特定位置打断点 可以精准快速的判断bug位置

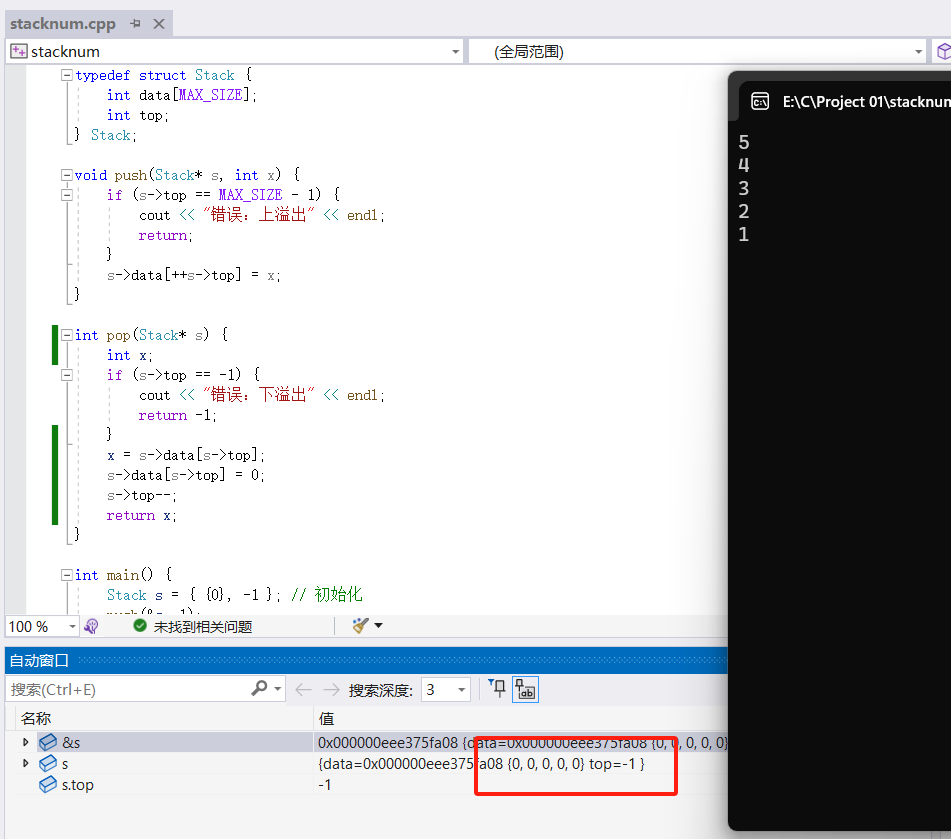


VS2022中还提供了强制运行执行到此处的功能

调试出栈函数 出栈函数需要确保返回的-1是一个错误值 不能被错误的使用

出栈需要恢复到初始值 链栈不需要考虑这个问题 直接将结点删去



修改对应代码

符合预期 使代码逻辑正确

#### 核心代码块：

typedef struct Stack {

int data[MAX\_SIZE];

int top;

} Stack;

void push(Stack\* s, int x) {

if (s->top == MAX\_SIZE - 1) {

cout << "错误：上溢出" << endl;

return;

}

s->data[++s->top] = x;

}

int pop(Stack\* s) {

int x;

if (s->top == -1) {

cout << "错误：下溢出" << endl;

return -1;

}

x = s->data[s->top];

s->data[s->top] = 0;

s->top--;

return x;

}

### 2.2.2 将栈的数据类型改为与问题匹配的Point

#### 运行和调试：

通过调试确保代码正确运行

#### 

#### 核心代码：

//修改栈中的元素类型 改为适合迷宫问题的Point 并且修改相应代码

// 定义Point结构体 包含坐标x，y

typedef struct Point {

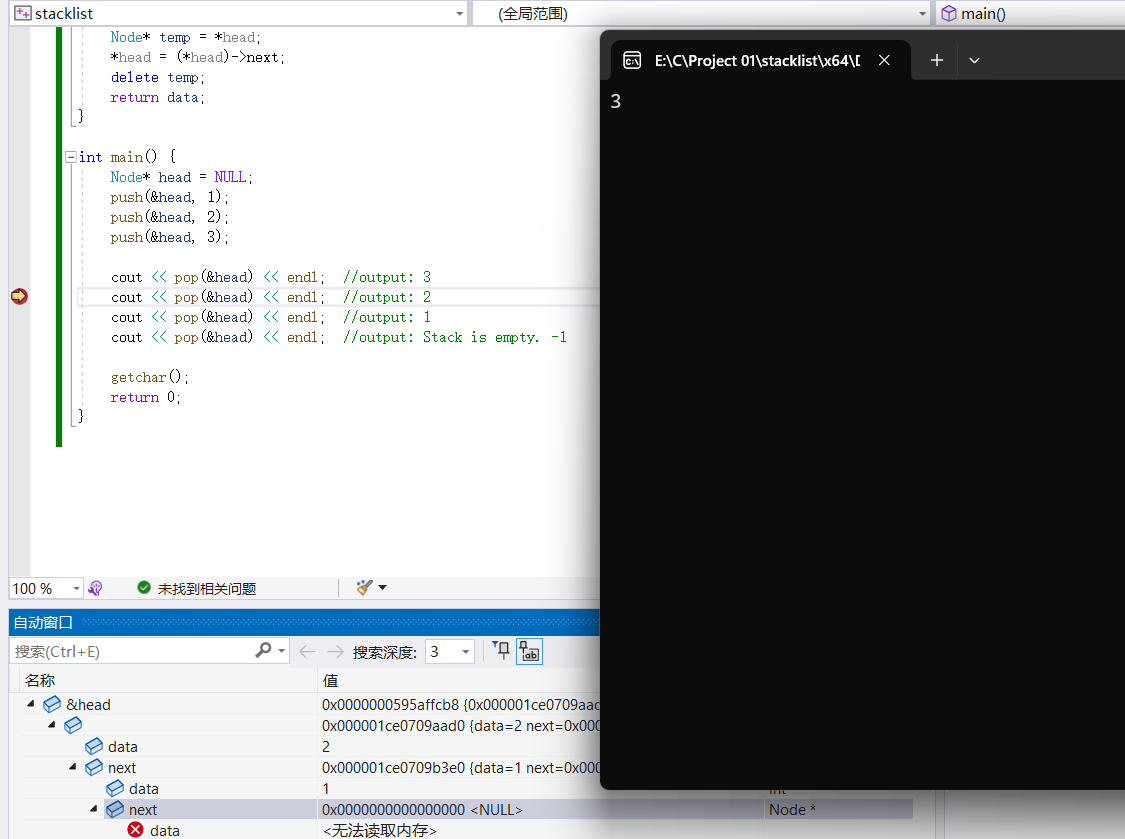
int x, y;

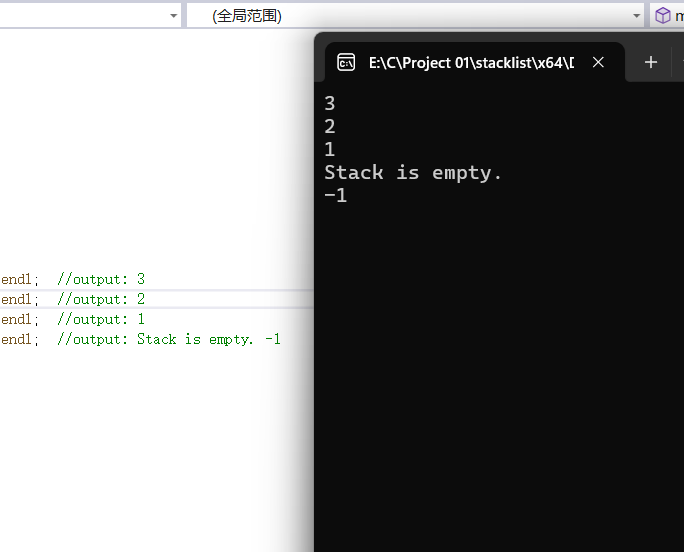
} Point;

### 2.2.3 链栈

#### 运行和调试：

将线性栈修改为链栈 通过调试确保代码正确运行



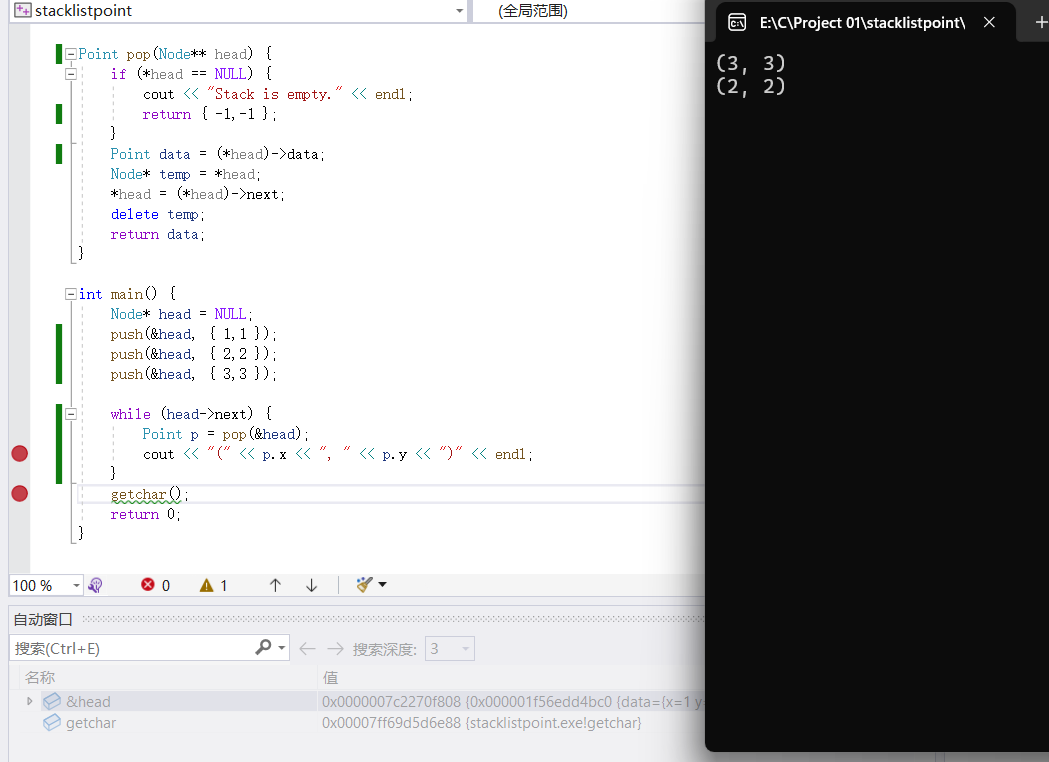
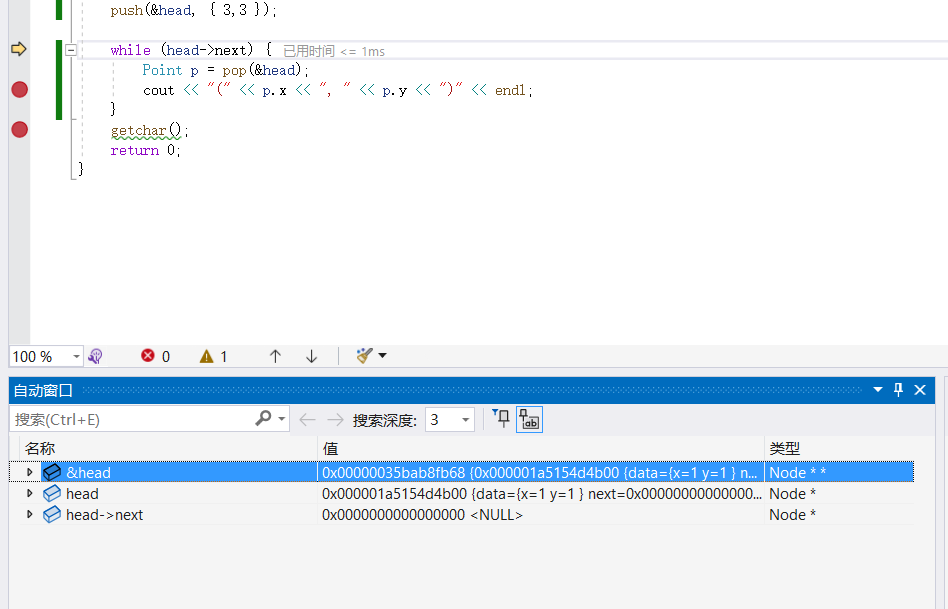
符合预期输出

### 2.2.4 Point类型的链栈

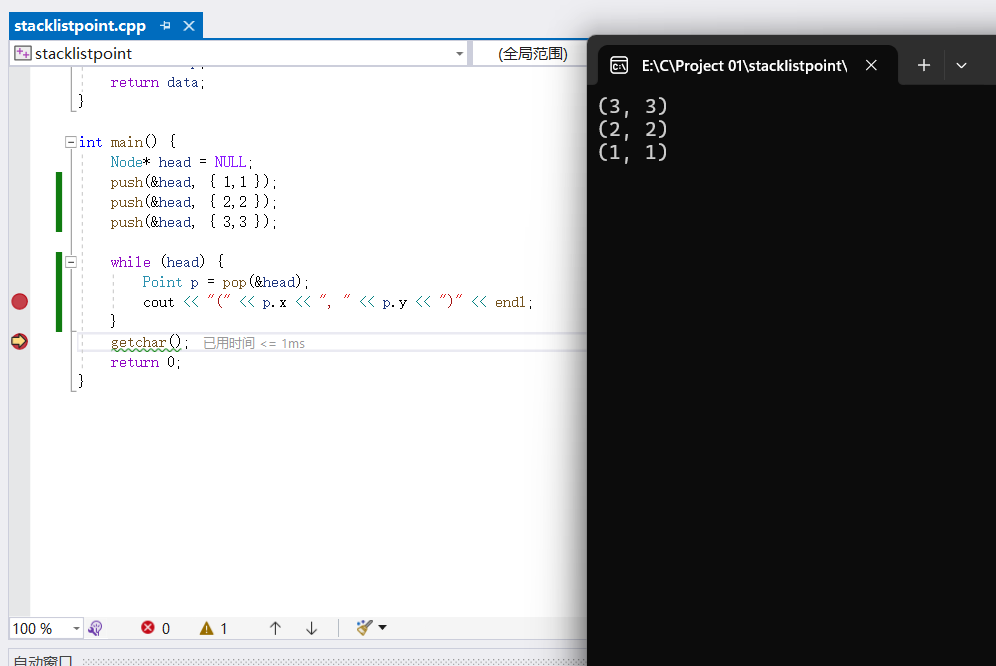
#### 运行和调试：

将结点类型改为适合迷宫问题的Point

在调试过程中发现 程序未能输出点(1,1)

通过逐语句调试发现问题在于while条件应该修改为head

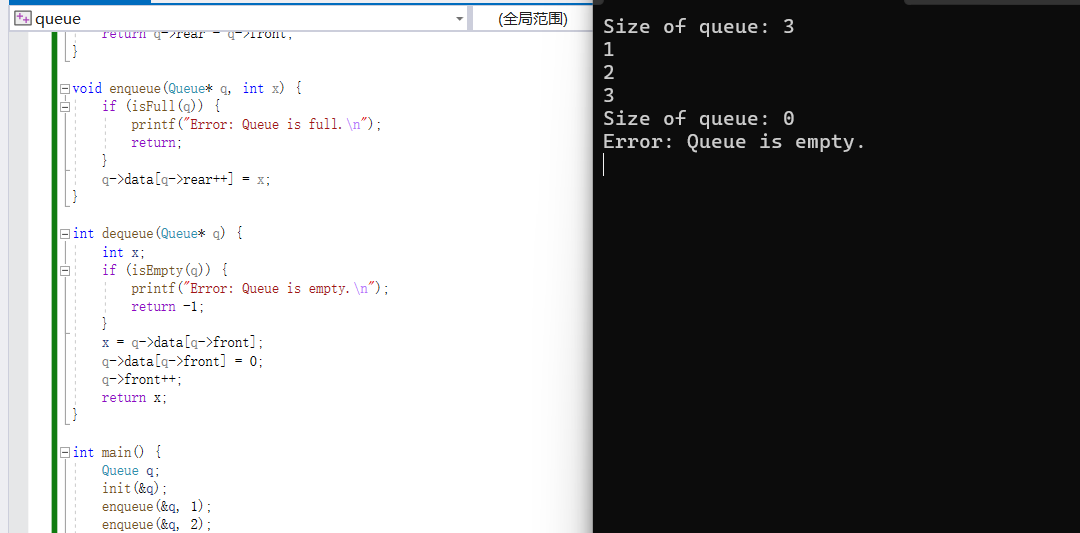
修改之后程序符合预期结果



### 2.2.5 队列

#### 运行和调试：

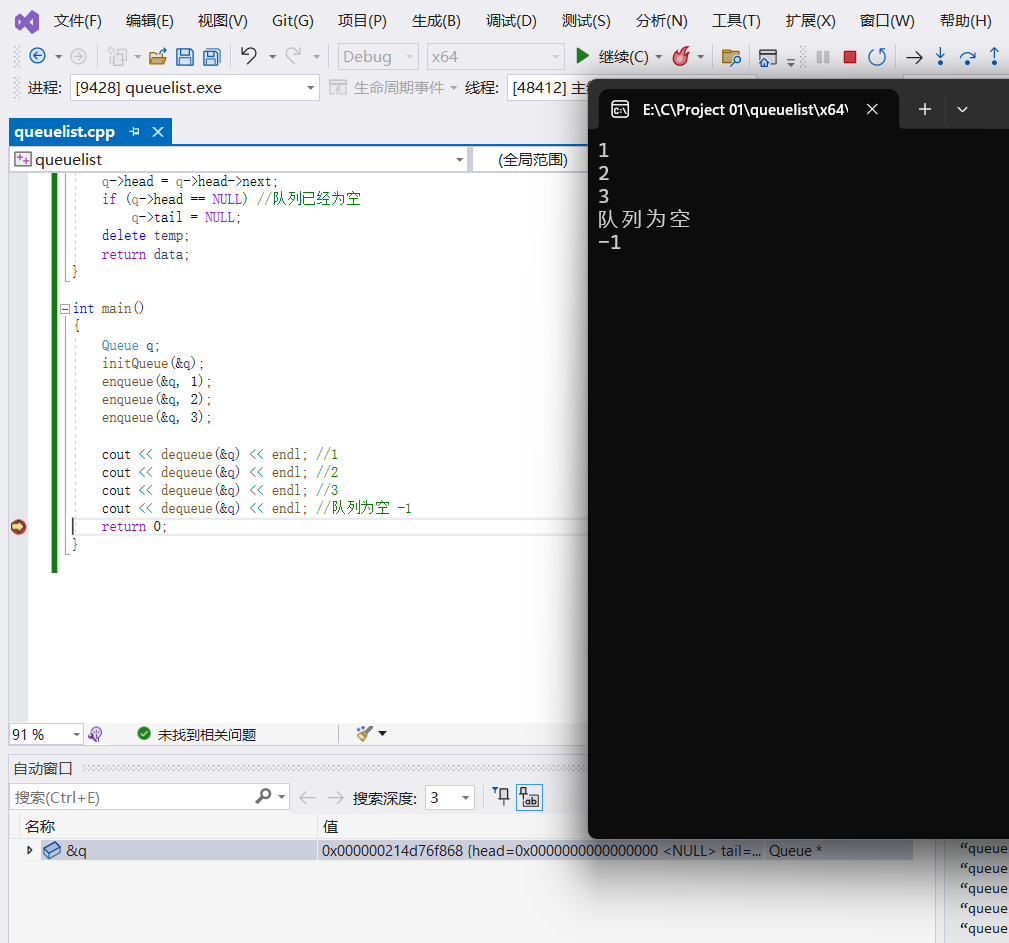
队列 先进先出 线性队列出队是注意重新赋为初始值 修改成Point类型与栈类似

这里不单独放调试结果

### 2.2.6 链队列

#### 运行和调试：

修改成链队列 并且完成调试



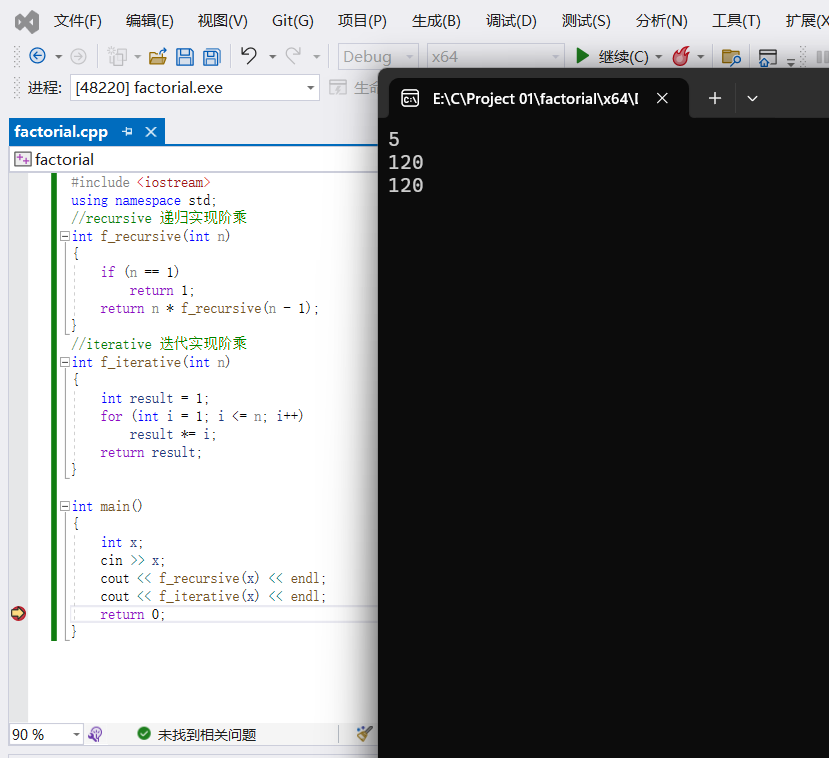
# 三、递归和迭代

## 3.1 理论

递归是通过函数调用自身来解决问题，它将问题分解成更小的子问题，直到达到一个基本条件为止。递归易于理解和实现，特别是在处理像树这样的数据结构时，但可能因为调用深度过深而导致内存消耗大和性能问题。

迭代利用循环结构重复执行操作直到满足特定条件，通常在性能上优于递归，因为它避免了额外的函数调用开销。迭代适用于大多数循环处理任务，能有效控制内存使用，但在解决某些问题时代码可能不如递归直观。

## 3.2 实验

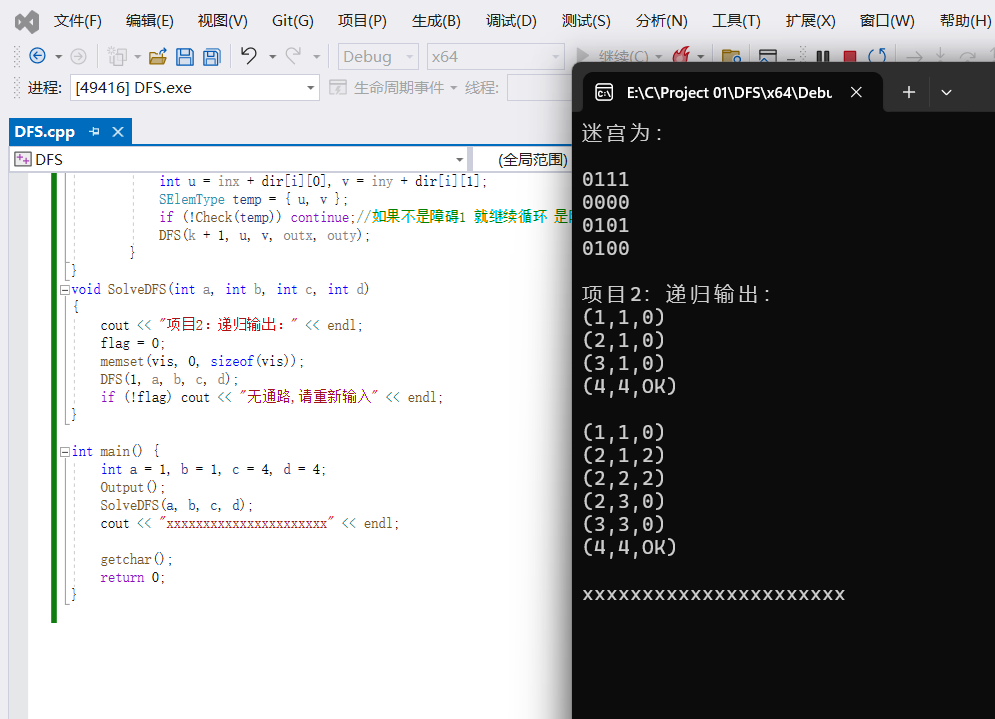
通过阶乘，比较递归和迭代的区别

# 四、迷宫问题

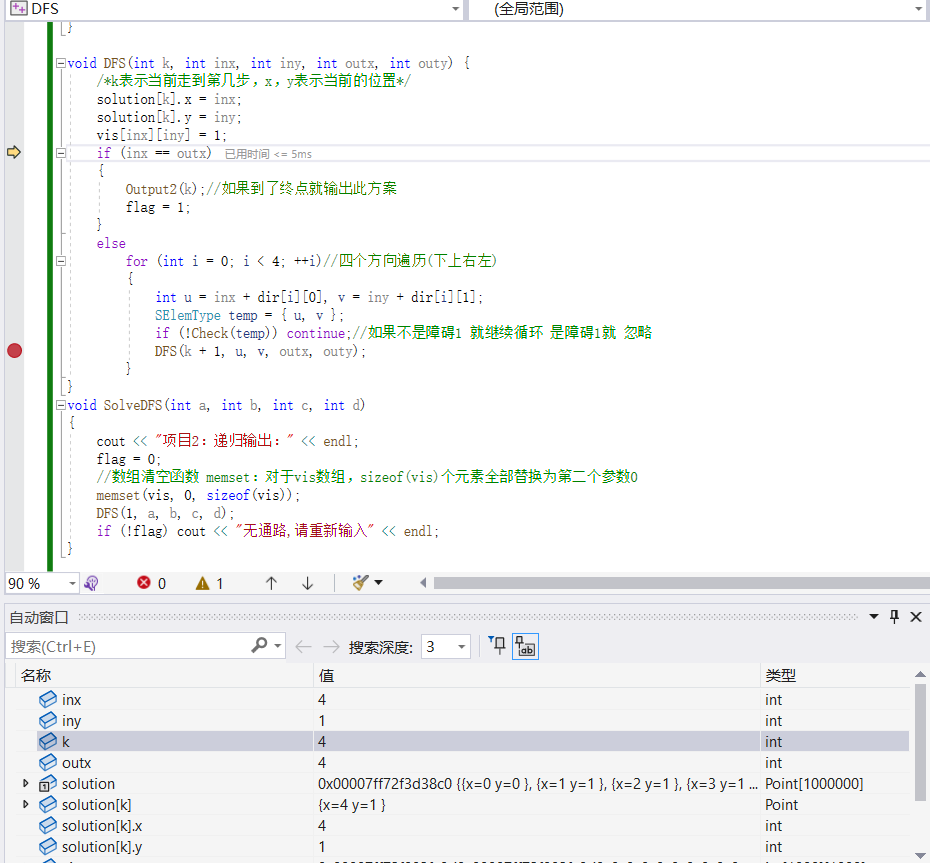
## 4.1 DFS递归实现

通过DFS递归实现迷宫问题

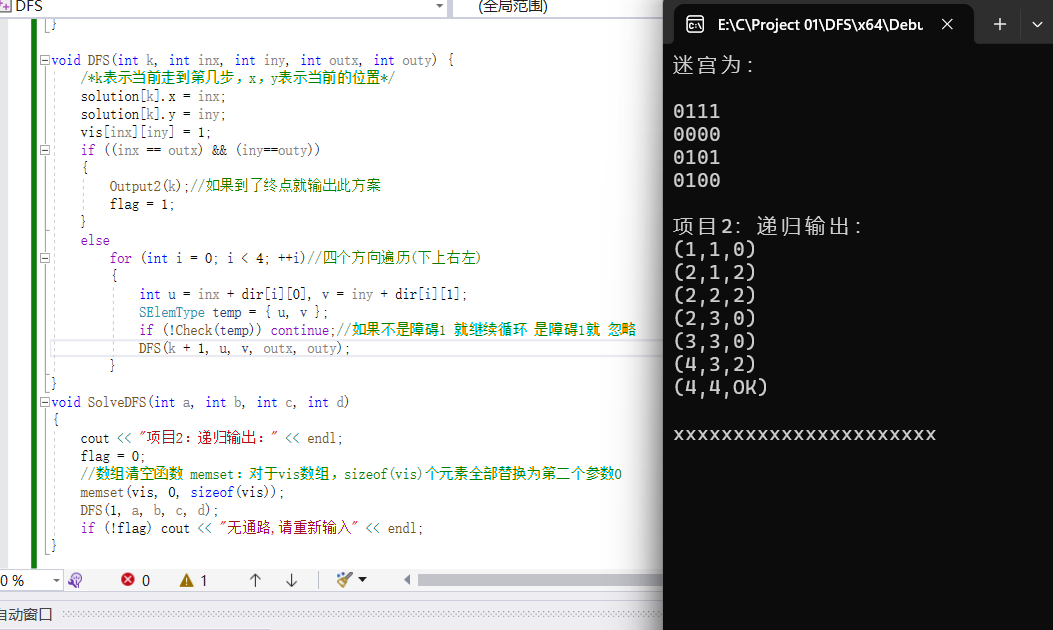
运行代码



发现并不满足实验要求 通过观察输出容易发现

观察发现第三个点处出现错误

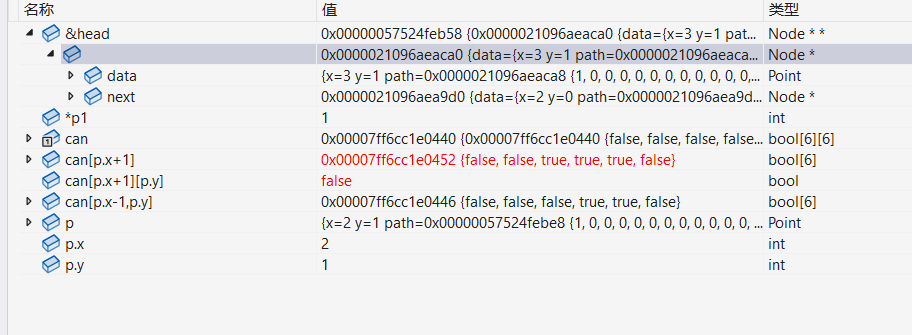
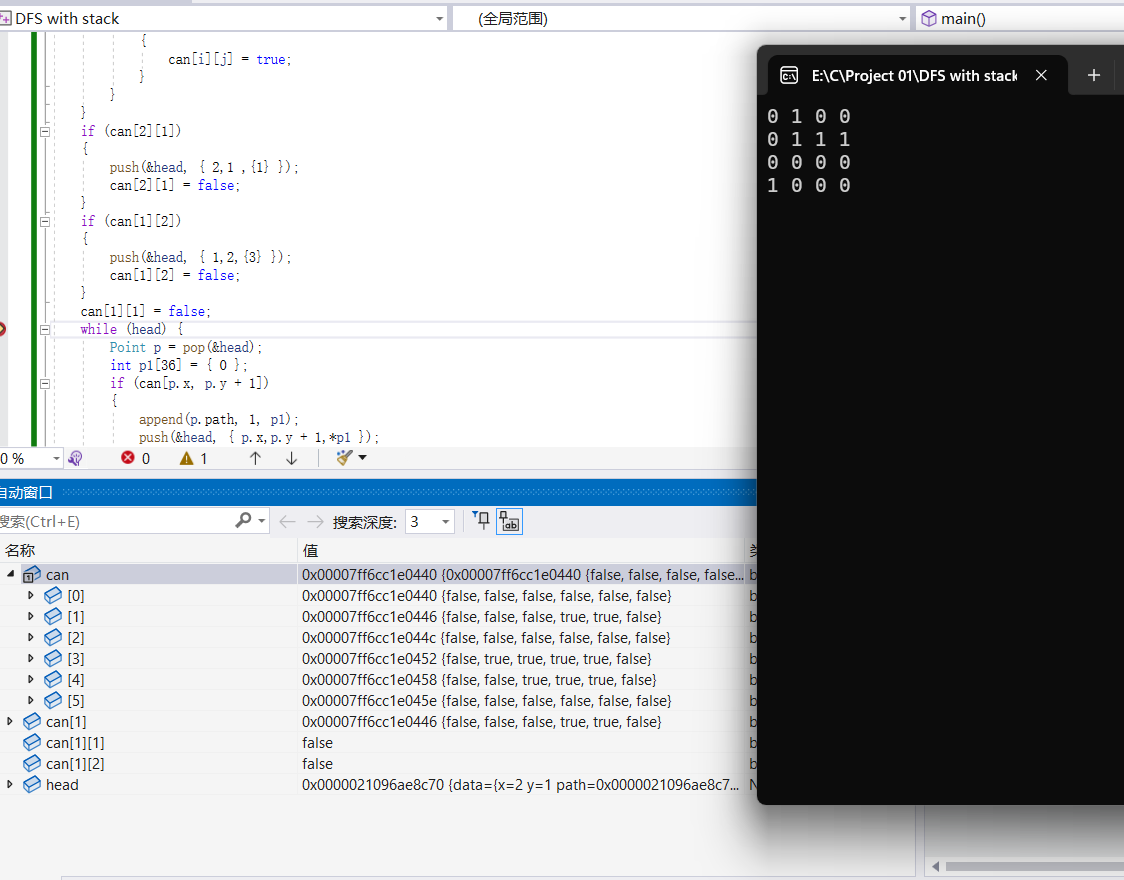
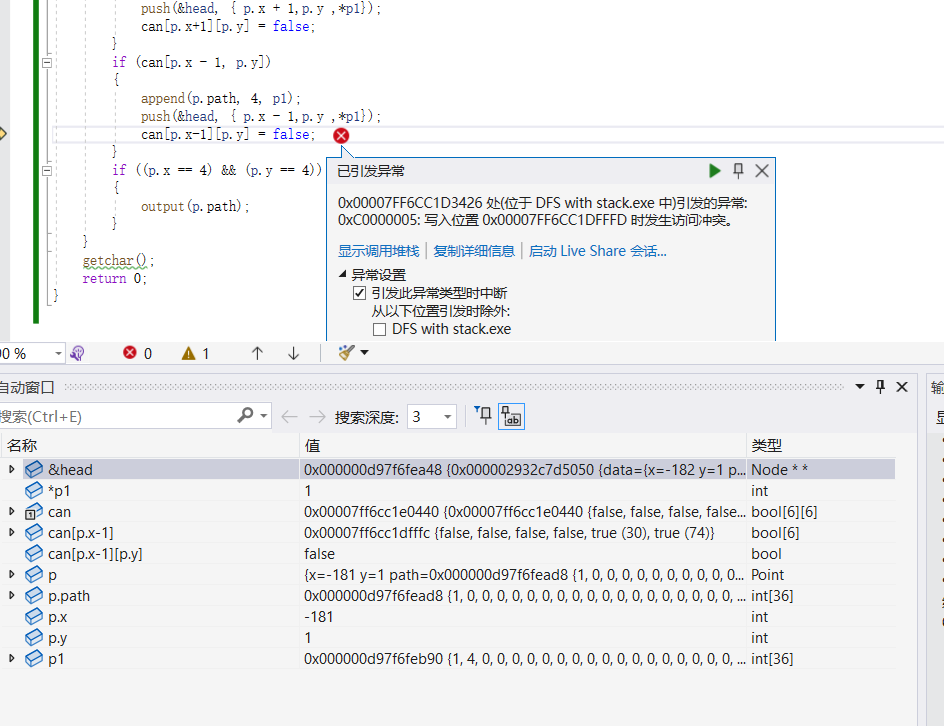
通过断点调试容易发现 在判断出口是只判断了x没有判断y



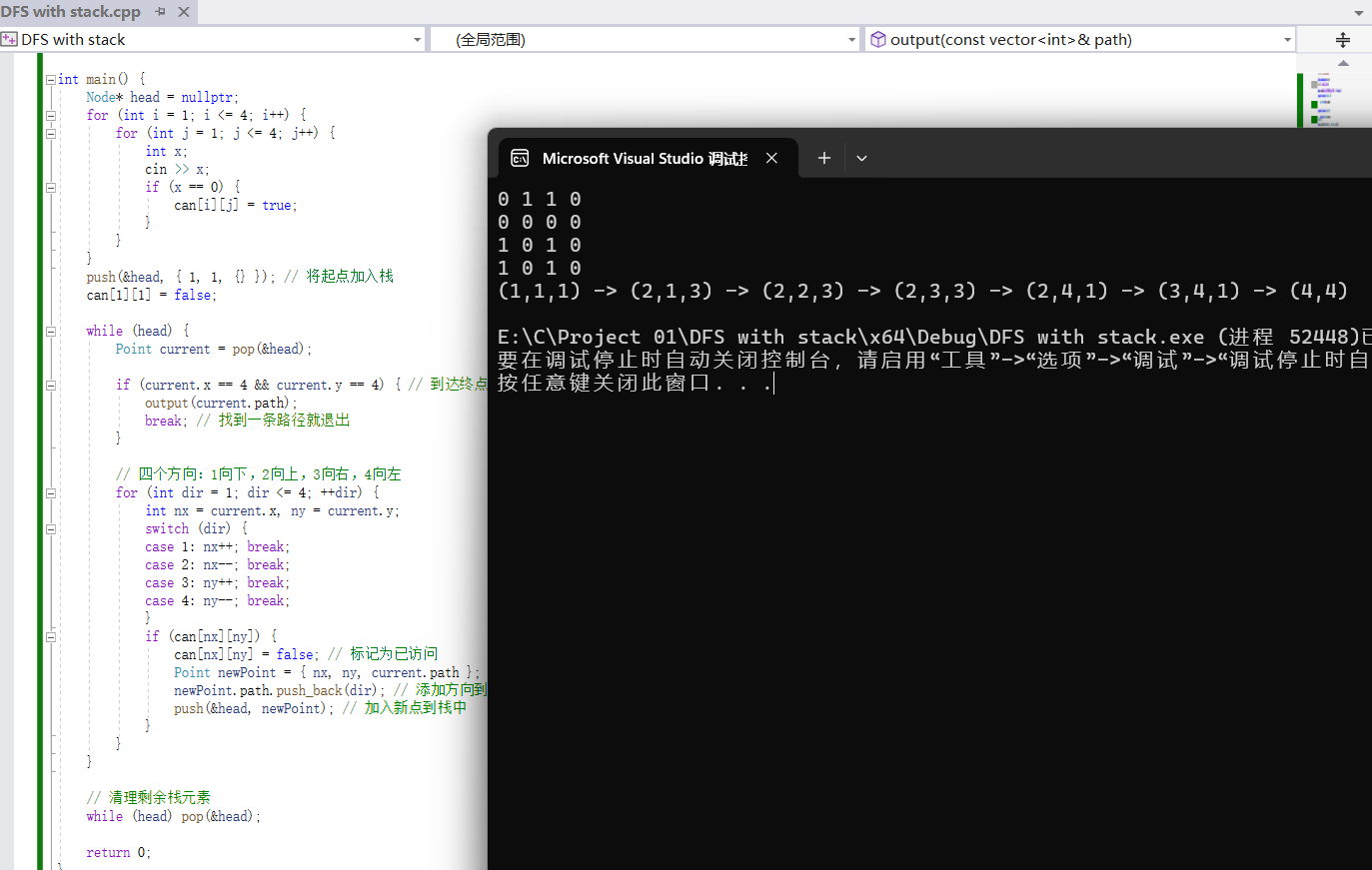
修改后发现运行正确

## 4.2 DFS迭代实现

编写用迭代方式实现深度优先搜索



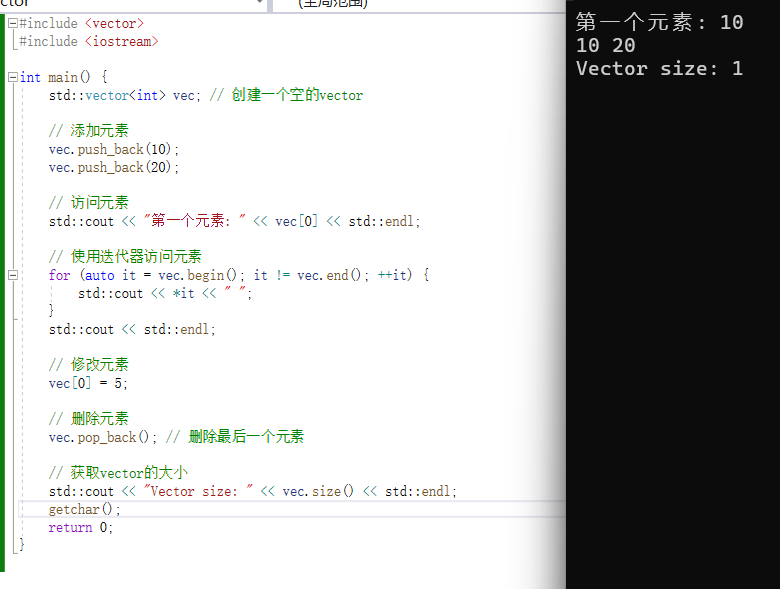
通过debug发现是由于搜索顺序导致的出错 由于在改代码基础上修改比较麻烦 我打算把代码重构 易知 在这个代码中path适用于动态数组类型 之后的代码中采用vector库 定义该数组 同时还修改了代码中的一些错误



修改后的代码符合实验预期要求 具体方向可见注释 需要注意的是 在程序的最后 应该清空栈 防止内存泄漏（C语言不会自动清除，内存泄漏的风险极大）

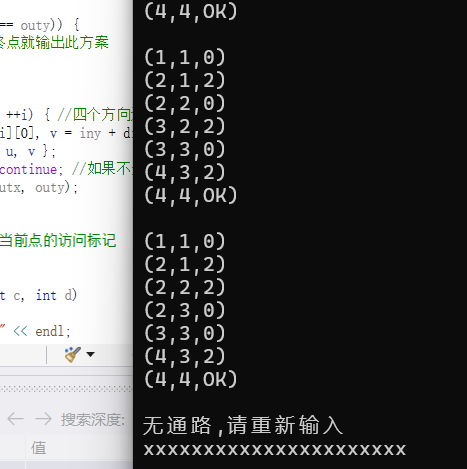
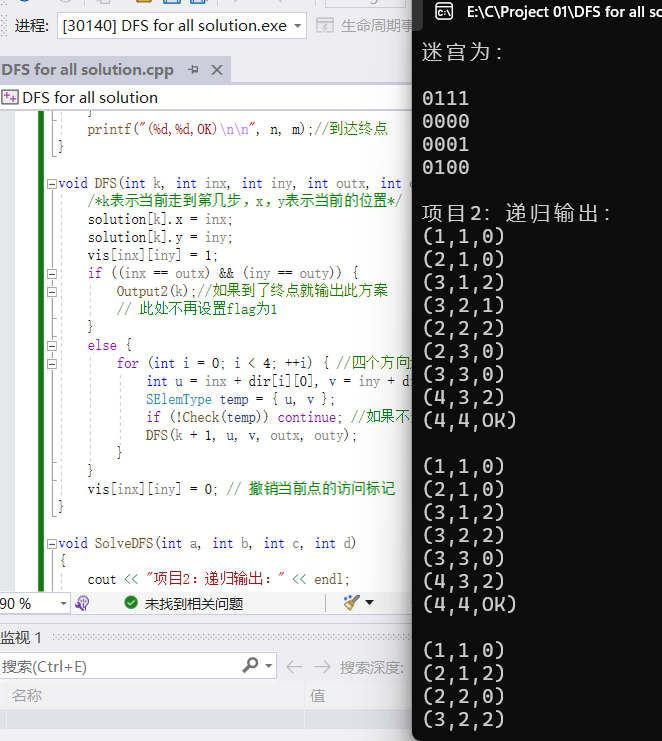
### 4.2（补充） Vector库介绍

vector 是 C++ 标准模板库（STL）中的一个非常重要且常用的容器类。它能够存储同一类型的动态数组，能够自动管理存储容量，当元素数量超出当前容量时，vector 会自动分配更大的存储空间，并将所有元素移动到新的存储空间中。

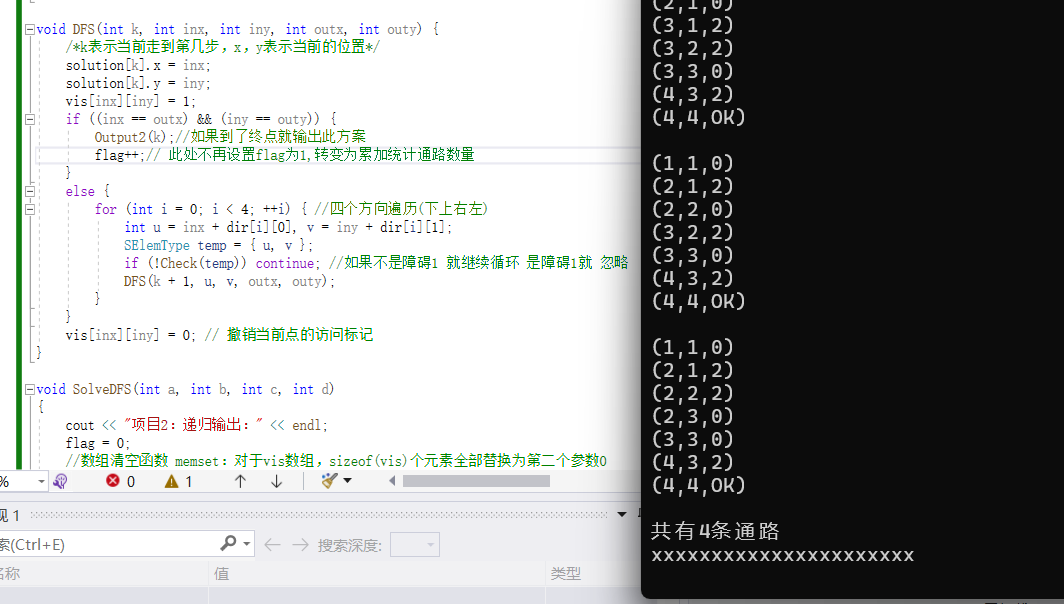


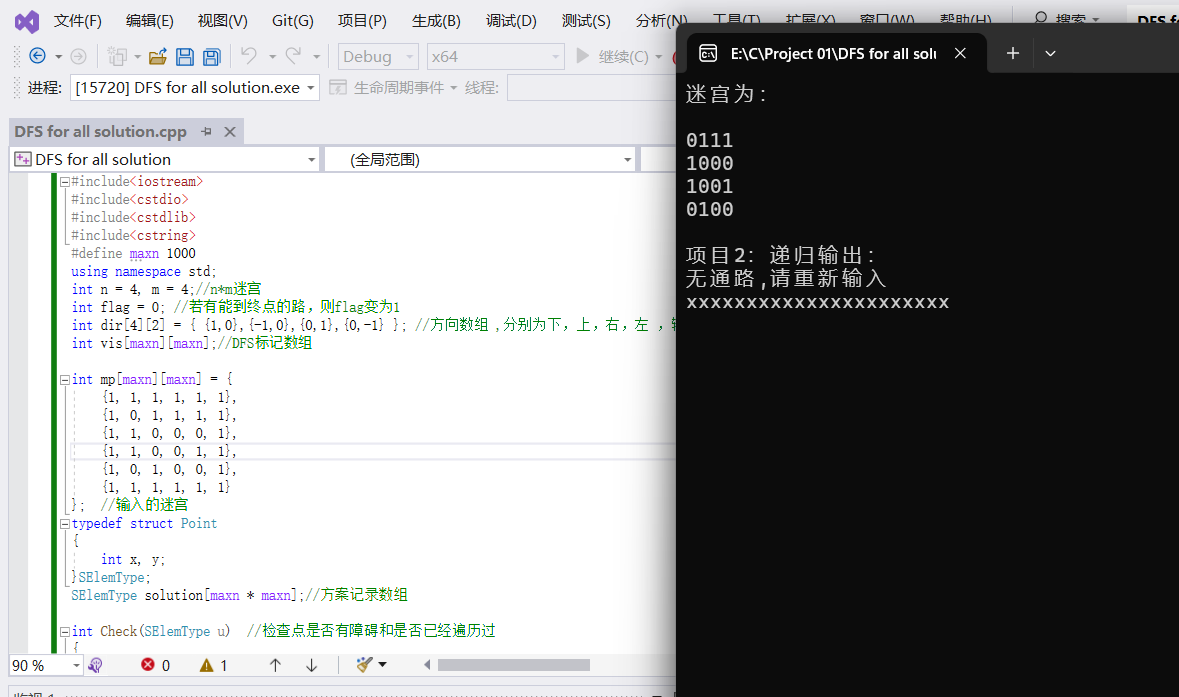
## 4.3 DFS输出所以通路

修改DFS代码 使得递归能输出所以通路 主要修改为撤销标记



我们需要假设 迷宫不能走回头路 通过观察发现 输出路线正确 但是最后仍输出了无通路，请重新输入 不满足实验预期 我们应该修改为输出路线有几条 如果是0 再输出：无通路，请重新输入





符合预期输出

#### 修改部分的核心代码：

void DFS(int k, int inx, int iny, int outx, int outy) {

/\*k表示当前走到第几步，x，y表示当前的位置\*/

solution[k].x = inx;

solution[k].y = iny;

vis[inx][iny] = 1;

if ((inx == outx) && (iny == outy)) {

Output2(k);//如果到了终点就输出此方案

flag++;// 此处不再设置flag为1,转变为累加统计通路数量

}

else {

for (int i = 0; i < 4; ++i) { //四个方向遍历(下上右左)

int u = inx + dir[i][0], v = iny + dir[i][1];

SElemType temp = { u, v };

if (!Check(temp)) continue; //如果不是障碍1 就继续循环 是障碍1就 忽略

DFS(k + 1, u, v, outx, outy);

}

}

vis[inx][iny] = 0; // 撤销当前点的访问标记

}}

## 4.4 BFS实现输出最短通路

用队列的方式编写BFS，并以方阵形式输出迷宫及其最短通路。

为了程序简洁和编写方便 这个代码直接采用了C++自带的queue库

在 C++ 标准库中，queue 是一种容器适配器，提供了先进先出 (FIFO) 数据结构的功能。它是在 <queue> 头文件中定义的，允许从队列的一端添加元素，并从另一端移除元素。queue 在 C++ 标准模板库 (STL) 中实现，它使用一个底层容器来存储元素。默认情况下，这个底层容器是 std::deque，也可以使用 std::list 或其他类型的容器。

主要操作

push()：向队尾添加一个元素。

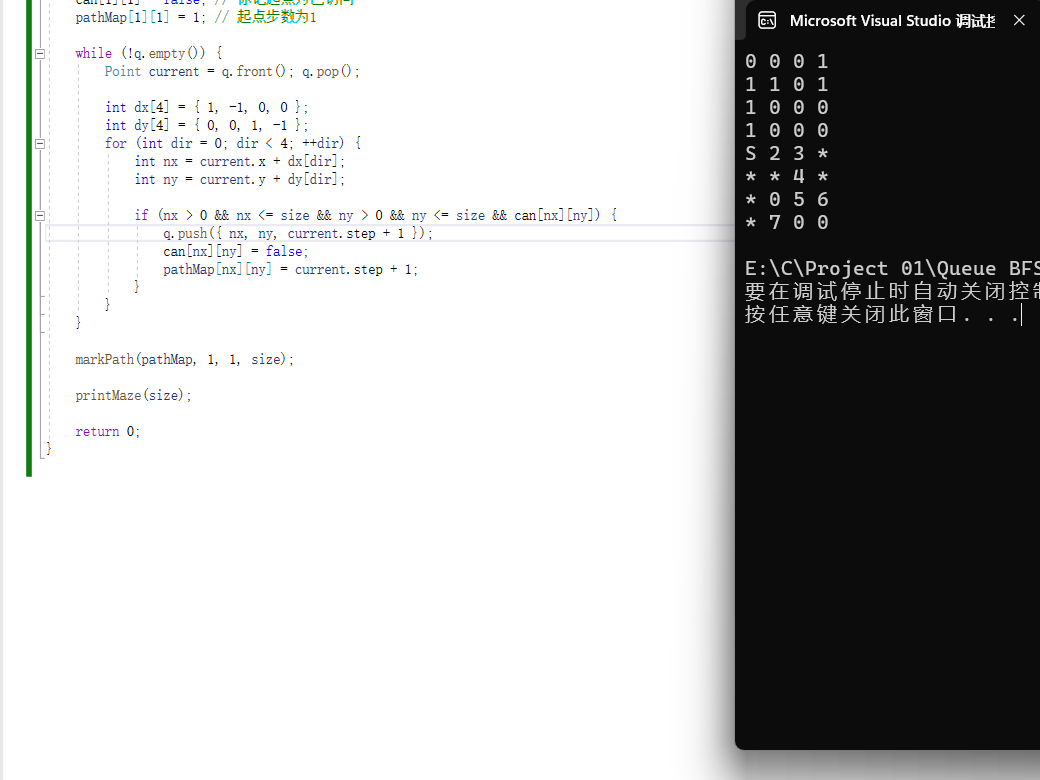
pop()：移除队首的元素。

front()：访问队首的元素。

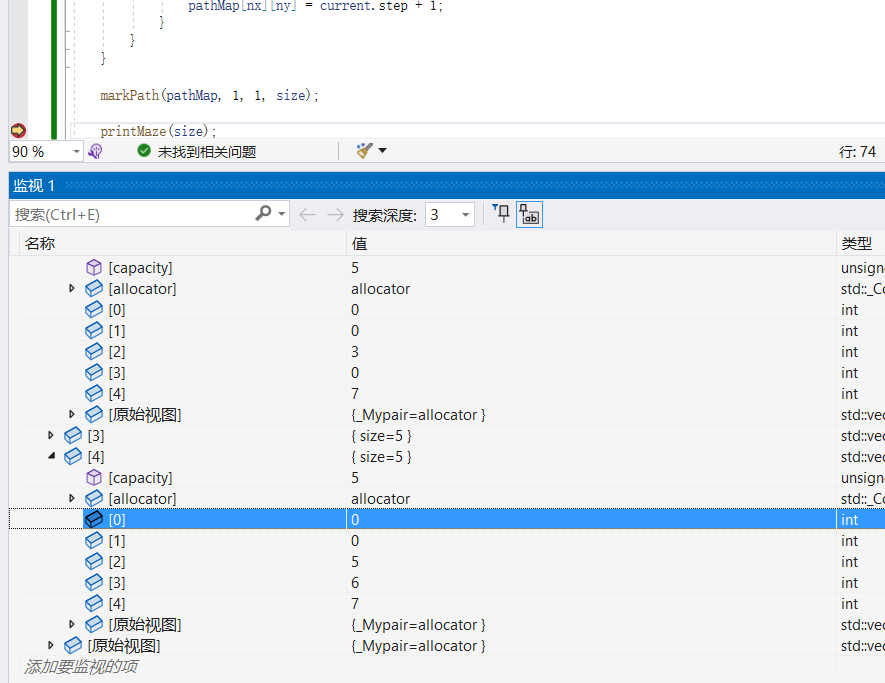
back()：访问队尾的元素。

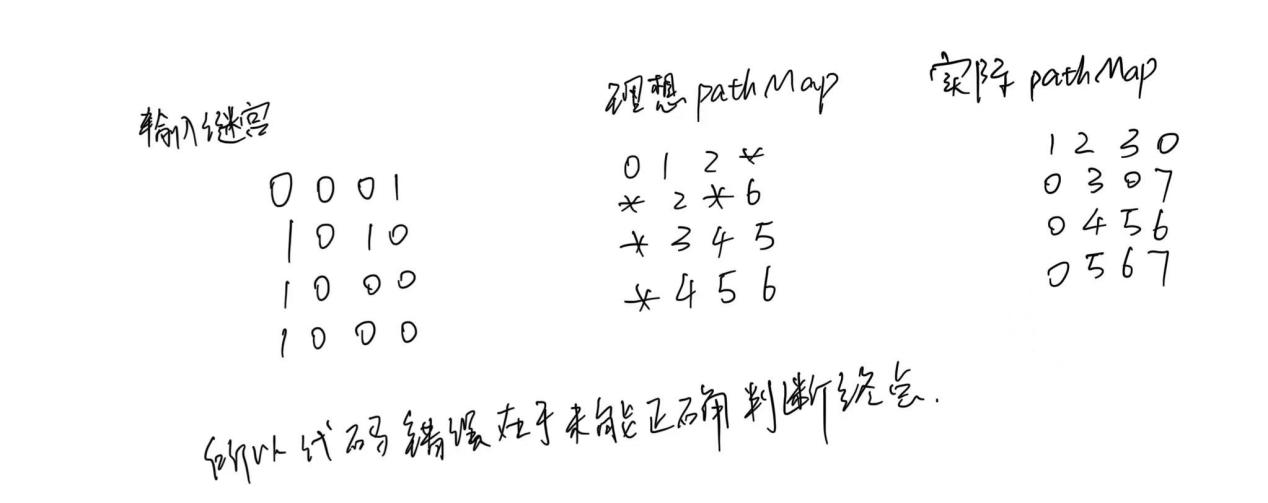
empty()：检查队列是否为空。

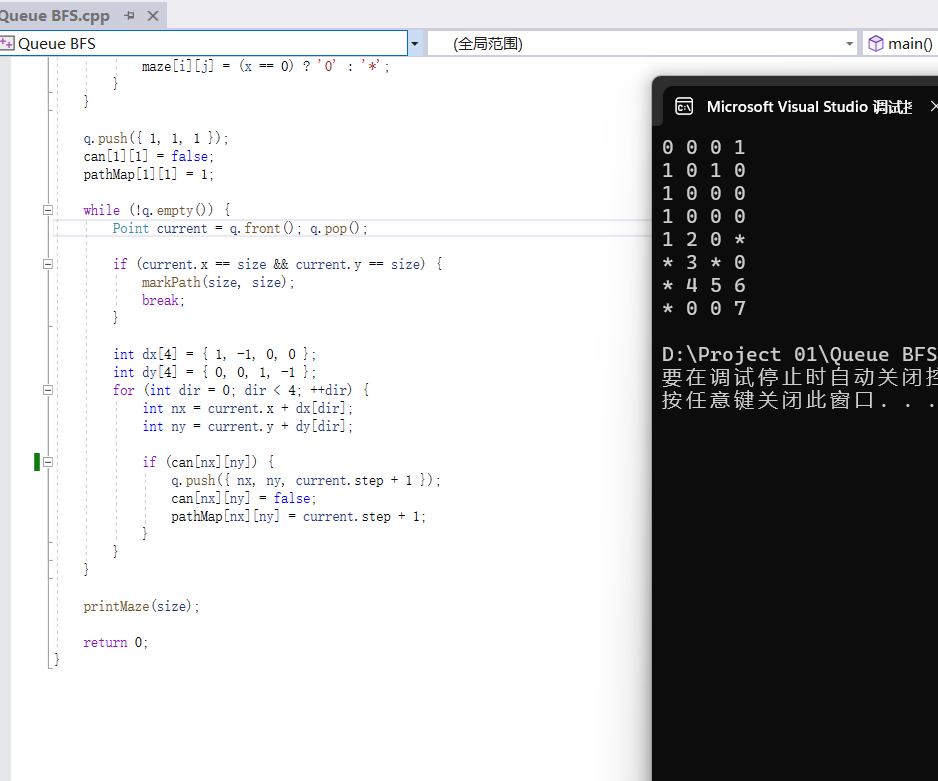
size()：返回队列中的元素个数。



完成编写后调试 发现未能按预期输出通路 最后的输出出现错误  
代码从逻辑上讲 只能记录一条通路 之后的会覆盖之前的路 这可能是导致输出错误的主要原因

于是通过监视判断pathMap是否有误.





在修改了代码之后 我确保了能输出一条最短的通路，实际上使用BFS确实可以找到所有最短路径，但实现方式相对复杂一些。为了收集所有最短路径，我们需要在每个点记录可能到达该点的所有前驱节点。然后，当我们到达终点时，通过回溯这些前驱节点来构建所有的路径。在这就不编写了。

## 4.5 DFS和BFS

### 4.5.1DFS和BFS的差异

在解决迷宫问题上，深度优先搜索（DFS）通过沿一条路径深入直到无法继续，然后回溯尝试其他路径，这样的策略使其在寻找所有可能解的过程中空间效率较高，尤其适合于解决路径深度较大的迷宫。然而，DFS的一个显著缺点是它可能找到的路径并不是最短的，甚至在不记录已访问节点的情况下可能陷入环路，使得搜索效率下降。

相反，广度优先搜索（BFS）通过逐层探索所有相邻节点，保证了每次扩展都是最短路径的增量，因此它能够保证找到从起点到终点的最短路径。这种策略使BFS在需要找到最短路径的迷宫问题中表现出色。然而，BFS的主要缺点是在宽度较大的迷宫中，需要维护的节点队列可能会非常庞大，导致空间复杂度显著增加。

解决迷宫问题时，旨在同时达到效率高和快速找到通路的目标，可以考虑使用启发式搜索算法，如A\*算法。A\*算法结合了BFS的优势（找到最短路径）和启发式信息（如欧几里得距离或曼哈顿距离作为估价函数），以评估从当前节点到目标节点的最佳路径。它通过计算每个节点的成本（实际从起点到该节点的成本加上该节点到终点的估计成本）来选择路径，这样既考虑了到达目标的实际成本，也利用了对目标距离的估计来指导搜索方向，避免无谓的探索。

**<https://blog.csdn.net/m0_59330466/article/details/129328732>**

### 4.5.2A\*算法

#### 4.5.2.1 A\*算法的优势

1.效率高：通过启发式函数指导搜索方向，减少探索不必要路径的次数，从而提高搜索效率。

2.找到最短路径：它保证在找到目标时，路径是最短的。

3.灵活性：启发式函数的选择可以根据具体问题进行调整，以适应不同的场景和需求。

在迷宫问题中，通过合理设计启发式函数，A\*算法能够有效地在大型迷宫中快速找到最短通路

#### 4.5.2.2 A\*算法伪代码

struct Node {

int x, y; // 节点在迷宫中的位置

int g; // 从起点到当前节点的成本

int h; // 启发式估计成本（当前节点到目标的估计成本）

int f; // f = g + h

struct Node\* parent; // 指向父节点的指针

};

// 假定有一个函数来计算h值，比如曼哈顿距离

int heuristic(Node\* start, Node\* goal) {

return abs(start->x - goal->x) + abs(start->y - goal->y);

}

// A\*搜索函数

void AStarSearch(Node\* start, Node\* goal) {

// 初始化开放列表和关闭列表

List openSet = createList();

List closedSet = createList();

// 将起点加入开放列表

addToList(openSet, start);

while (!isListEmpty(openSet)) {

// 在开放列表中查找具有最低f值的节点作为当前节点

Node\* current = findLowestF(openSet);

// 如果当前节点是目标，重建路径并返回

if (current == goal) {

reconstructPath(current);

return;

}

// 将当前节点从开放列表移除并加入关闭列表

removeFromList(openSet, current);

addToList(closedSet, current);

// 遍历当前节点的所有邻居

foreach(neighbor in getNeighbors(current)) {

if (isInList(closedSet, neighbor)) {

continue; // 如果邻居在关闭列表中，跳过

}

// 计算从起点经当前节点至邻居的成本

int tentative\_gScore = current->g + distance(current, neighbor);

if (!isInList(openSet, neighbor)) {

addToList(openSet, neighbor); // 发现一个新节点

}

else if (tentative\_gScore >= neighbor->g) {

continue; // 这不是一个更好的路径

}

// 记录最佳路径到目前为止

neighbor->parent = current;

neighbor->g = tentative\_gScore;

neighbor->h = heuristic(neighbor, goal);

neighbor->f = neighbor->g + neighbor->h;

}

}

// 如果开放列表被耗尽仍未找到路径，返回失败

return failure;

}

# 五、实验总结

这次实验内容相对容易，主要复习了BFS和DFS，栈和队列，迭代和递归三个有联系又有区别算法和数据结构。同时学习了使用VS2022对代码进行Debug，在实验过程中遇到一些问题，但是通过Debug能够使问题基本解决，锻炼了我写代码，尤其是栈和队列的数据结构，同时锻炼了我调试代码的能力。

在进行扩展任务时，能够顺利完成BFS算法的编写，同时通过对比DFS和BFS ，联想到A\*算法可以快速的寻找到最优路径，虽然没有完成代码的编写，但是通过查询资料，编写了一部分伪代码，A\*算法在人工智能的应用，也会为我之后的学习带来颇多的收获。