

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
1.1 实验目的	3
1.2 实验内容	3
1.3 实验要求	3
1.4 实验任务	3
二、栈和队列	4
2.1 问题描述	4
2.2 栈和队列	4
2.2.1 栈	4
2.2.2 将栈的数据类型改为与问题匹配的 Point	9
2.2.3 链栈	10
2.2.4 Point 类型的链栈	12
2.2.5 队列	14
2.2.6 链队列	14
三、递归和迭代	15
3.1 理论	15
3.2 实验	16
四、迷宫问题	16
4.1 DFS 递归实现	16
4.2 DFS 迭代实现	18
4.2 (补充) Vector 库介绍	20
4.3 DFS 输出所有通路	21
修改部分的核心代码	24
4.4 BFS 实现输出最短通路	24
4.5 DFS 和 BFS	27
4.5.1 DFS 和 BFS 的差异	27
4.5.2 A*算法	28
五、实验总结	30

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 $m \times n$ 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 $m \times n$ 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:

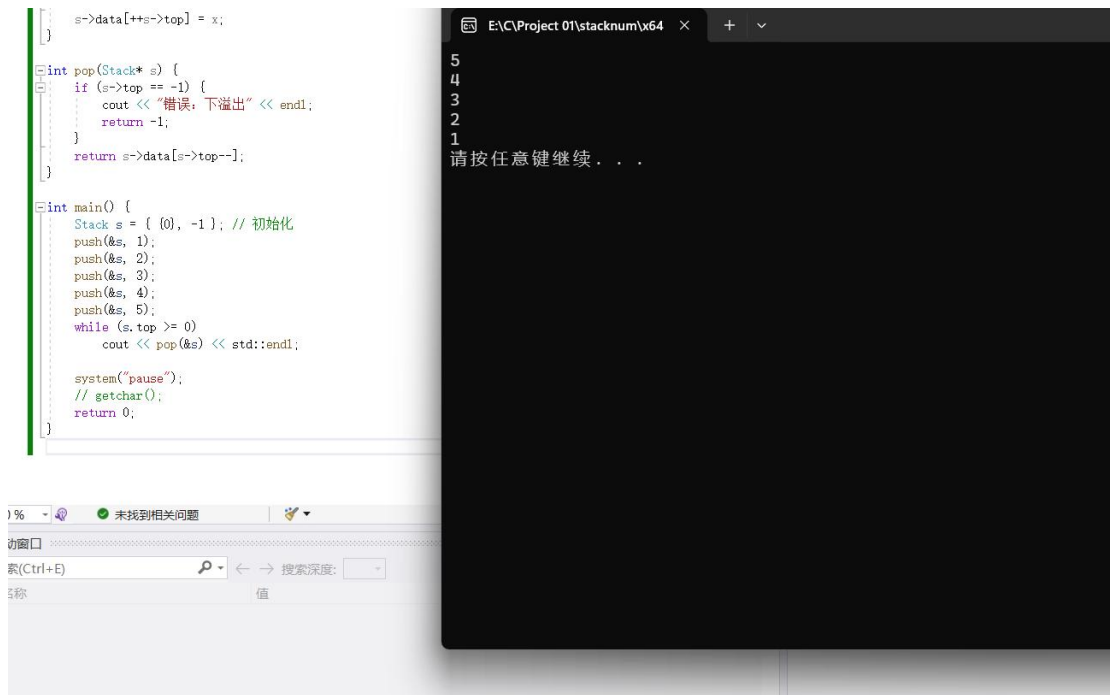
1. 创建一个空栈, 并将起点压入栈中。
2. 当栈不为空时, 判断栈顶元素, 如果该元素为终点, 则搜索结束, 否则将其标记为已访问。
3. 检查当前结点的相邻结点是否可行 (未被访问且不是障碍物), 如果可行则将其压入栈中; 若不可行则将该结点出栈。
4. 重复步骤 2-3, 直到找到终点或栈为空。
5. 如果栈为空且未找到终点, 则无解。

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

```
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;
    已用时间 <= 5ms
}

int pop(Stack* s) {
    if (s->top == -1) {
        cout << "错误: 下溢出" << endl;
        return -1;
    }
    return s->data[s->top--];
}

int main() {
    Stack s = { {0}, -1 }; // 初始化
    push(&s, 1);
    push(&s, 2);
    push(&s, 3);
    push(&s, 4);
}
```

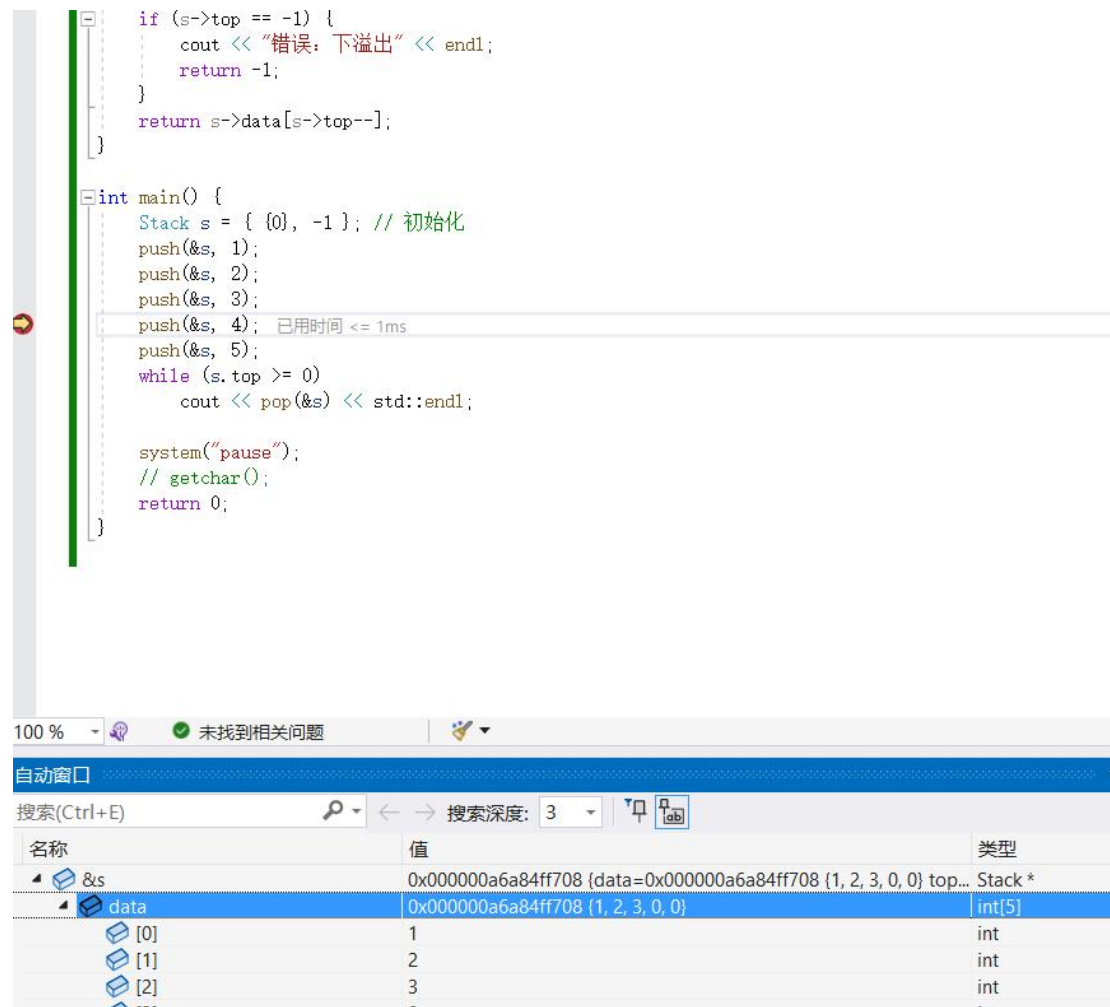
100 % 未找到相关问题

自动窗口

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
s	0x000000a6a84ff708 {data=0x000000a6a84ff708 {1, 2, 0, 0, 0} top...	Stack *
s->data	0x000000a6a84ff708 {1, 2, 0, 0, 0}	int[5]
s->data[s->top]	2	int
s->top	1	int
x	2	int

可以用逐语句调试观察到栈内元素变化

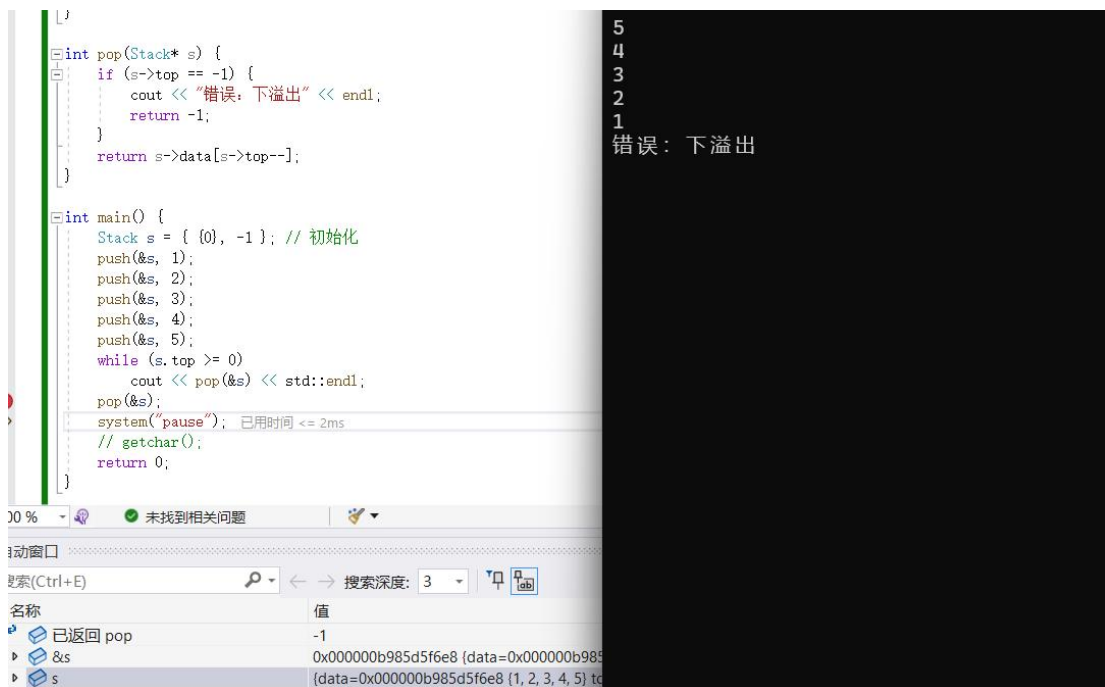


也可以用断点的形式呈现 通过在特定位置打断点 可以精准快速的判断 bug 位置

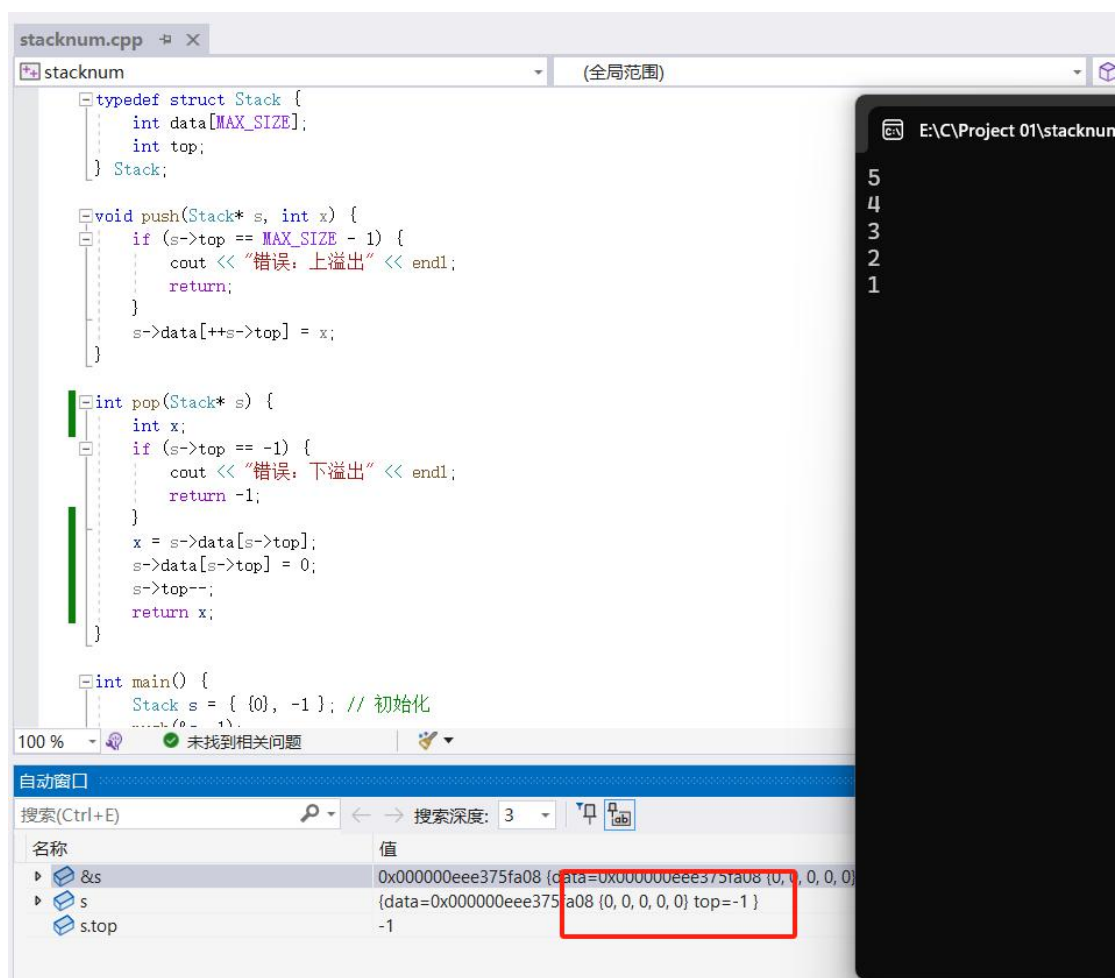


VS2022 中还提供了强制运行执行到此处的功能

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;

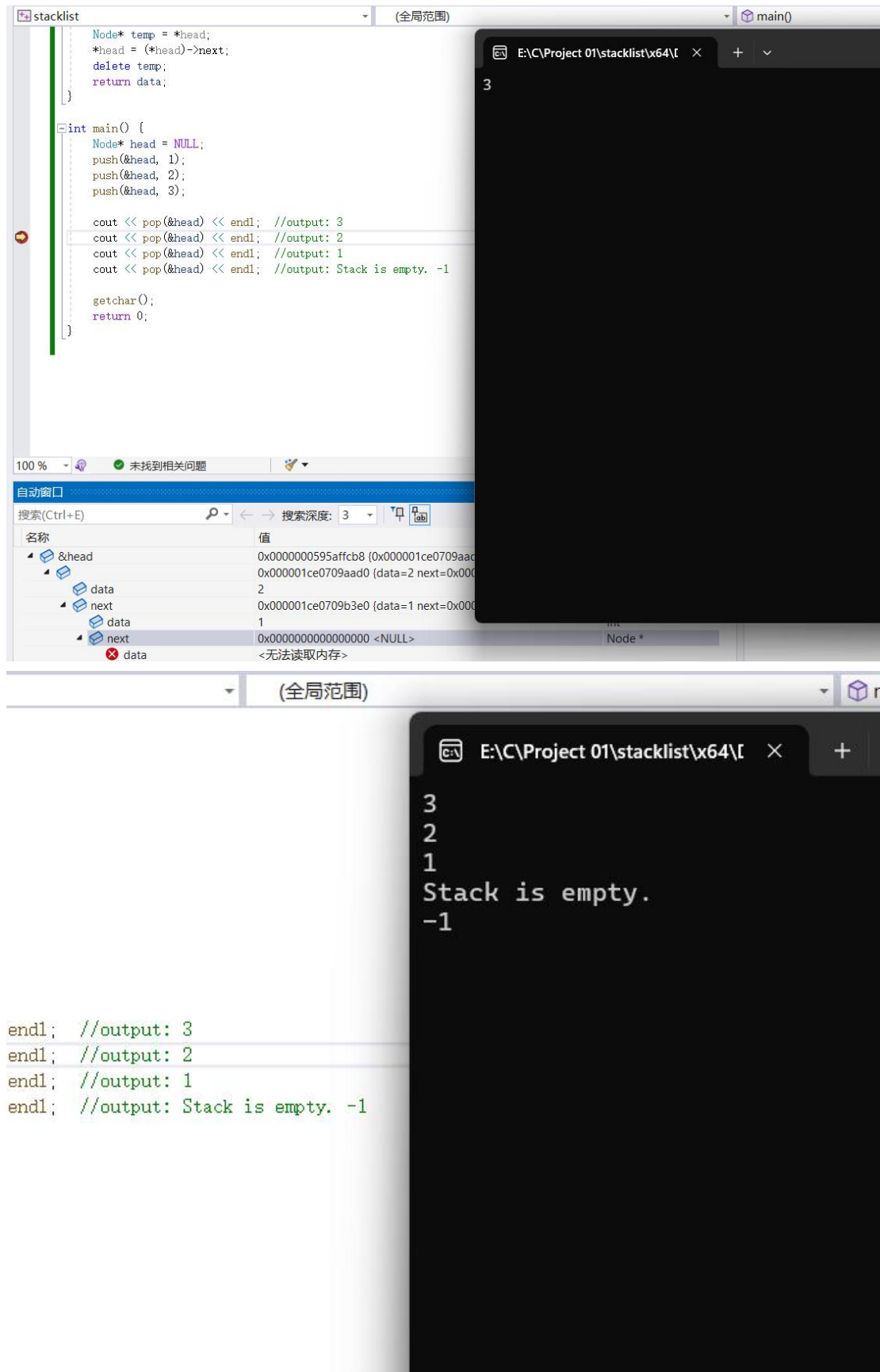
void push(Stack* s, int x) {
    if (s->top == MAX_SIZE - 1) {
        cout << "错误：上溢出" << endl;
        return;
    }
    s->data[++s->top] = x;
}

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



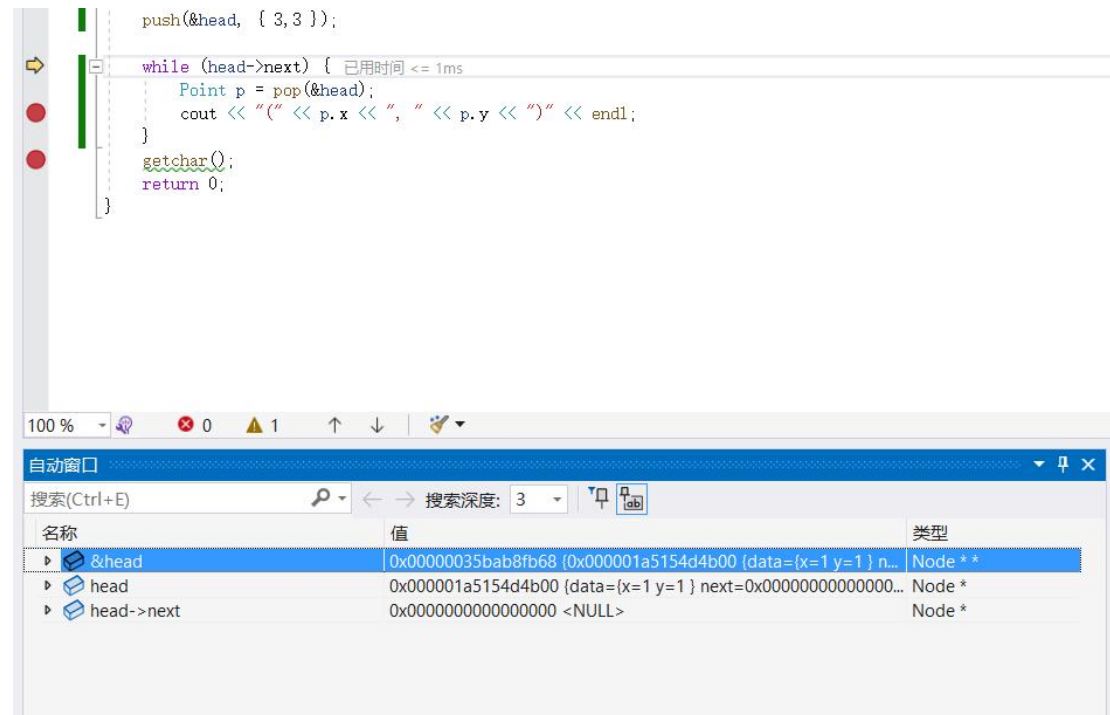
符合预期输出

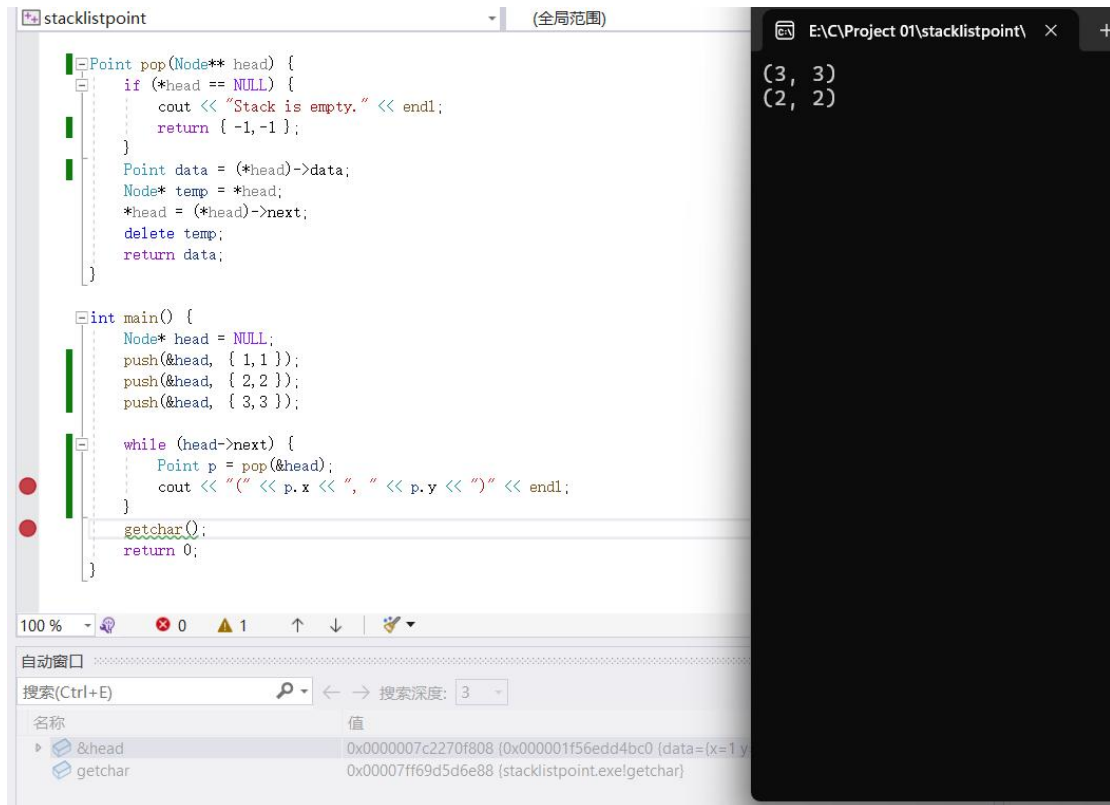
2.2.4 Point 类型的链栈

运行和调试：

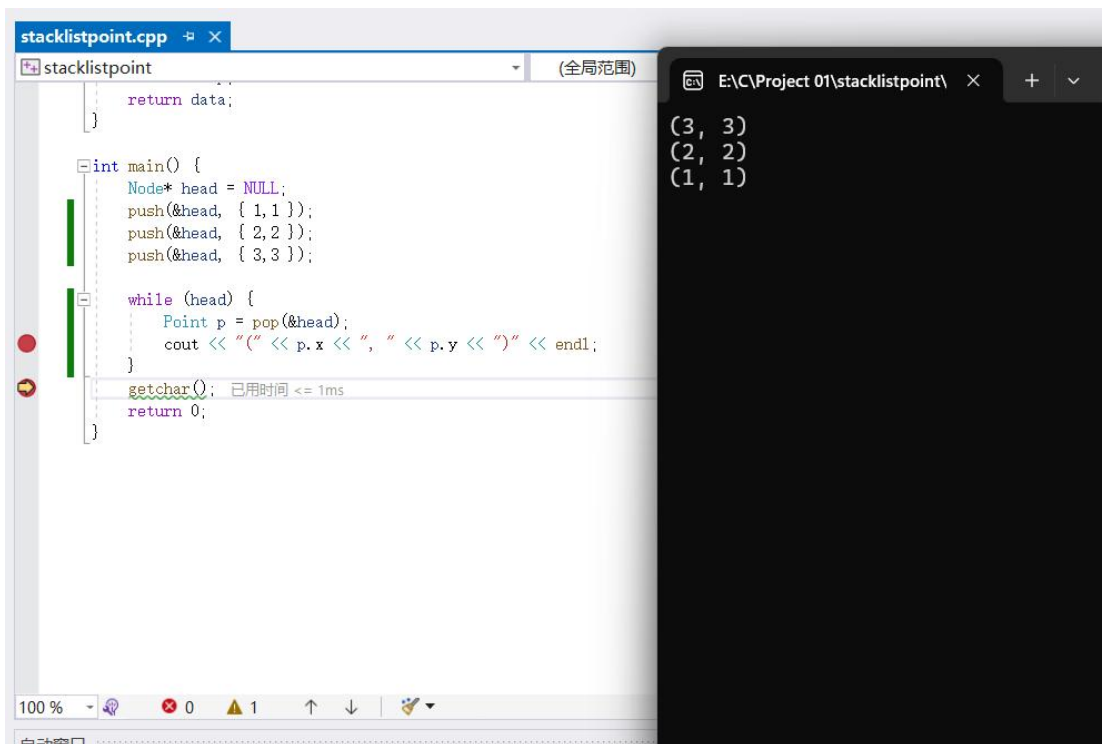
将结点类型改为适合迷宫问题的 Point

在调试过程中发现 程序未能输出点(1,1)





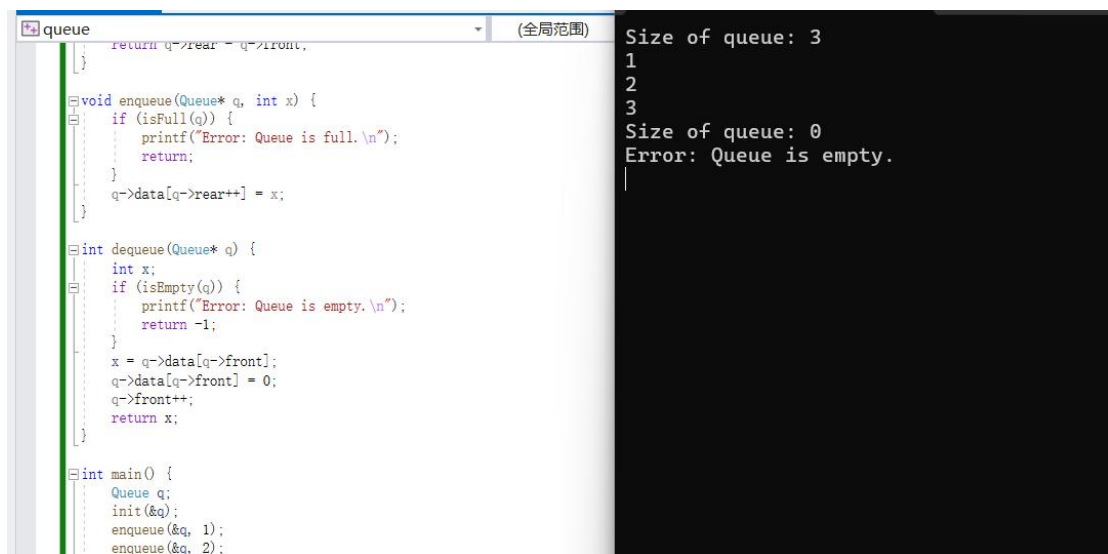
通过逐语句调试发现问题在于 while 条件应该修改为 head
修改之后程序符合预期结果



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



The screenshot shows a C++ IDE with a file named 'queue'. The code implements a queue using an array. The 'enqueue' function checks if the queue is full before adding an element. The 'dequeue' function checks if the queue is empty before removing an element. The 'main' function initializes the queue and enqueues the values 1 and 2. The output window on the right shows the execution results.

```
return q->rear - q->front;
}

void enqueue(Queue* q, int x) {
    if (isFull(q)) {
        printf("Error: Queue is full.\n");
        return;
    }
    q->data[q->rear++] = x;
}

int dequeue(Queue* q) {
    int x;
    if (isEmpty(q)) {
        printf("Error: Queue is empty.\n");
        return -1;
    }
    x = q->data[q->front];
    q->data[q->front] = 0;
    q->front++;
    return x;
}

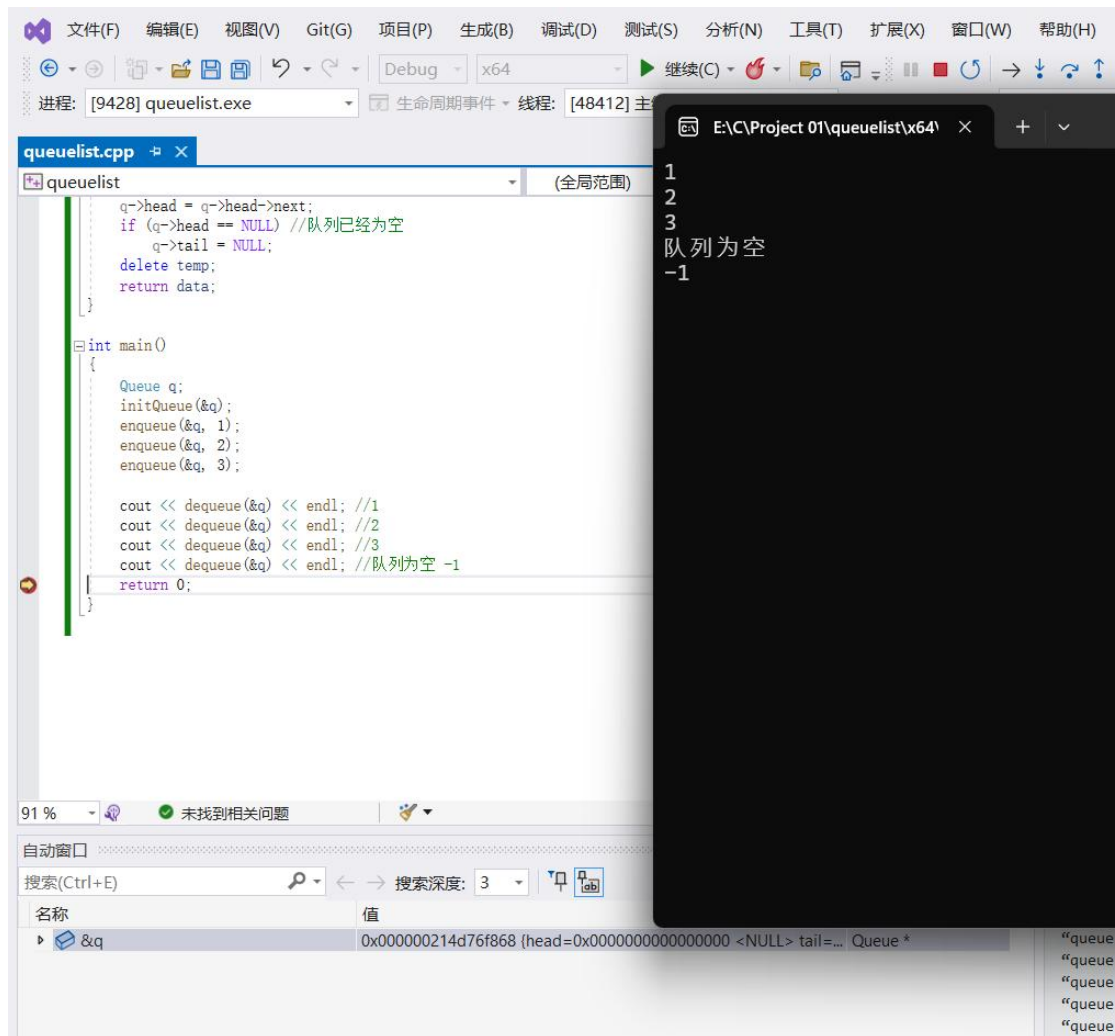
int main() {
    Queue q;
    init(&q);
    enqueue(&q, 1);
    enqueue(&q, 2);
}
```

Size of queue: 3
1
2
3
Size of queue: 0
Error: Queue is empty.

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

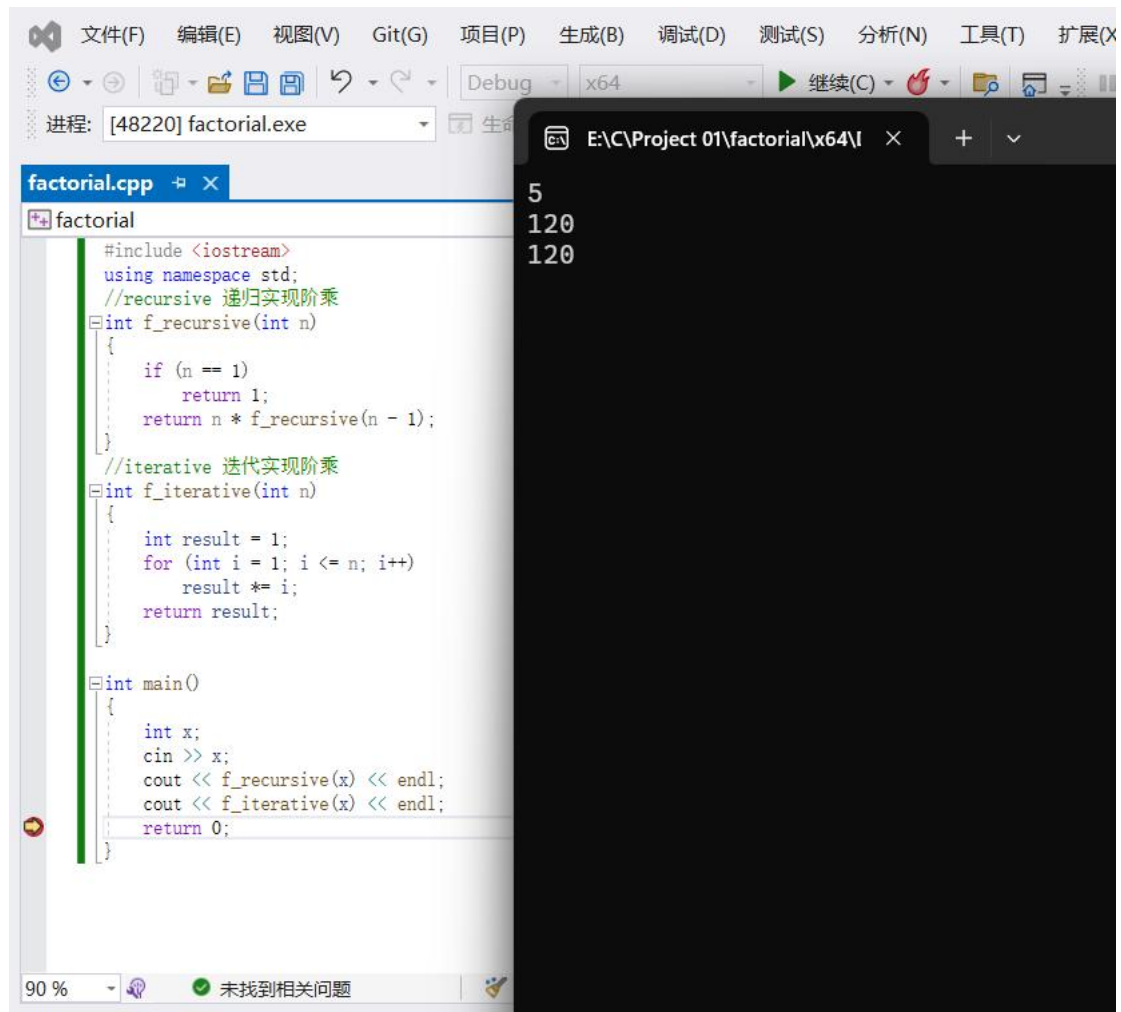
3.1 理论

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



The screenshot shows a C++ IDE with a file named `factorial.cpp` open. The code implements two functions to calculate factorials: `f_recursive` (recursive) and `f_iterative` (iterative). The `main` function reads an integer `x` from the user and prints the results of both functions. The output window on the right shows the input `5` and the corresponding recursive and iterative results, both of which are `120`.

```
#include <iostream>
using namespace std;
//recursive 递归实现阶乘
int f_recursive(int n)
{
    if (n == 1)
        return 1;
    return n * f_recursive(n - 1);
}
//iterative 迭代实现阶乘
int f_iterative(int n)
{
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main()
{
    int x;
    cin >> x;
    cout << f_recursive(x) << endl;
    cout << f_iterative(x) << endl;
    return 0;
}
```

Output:

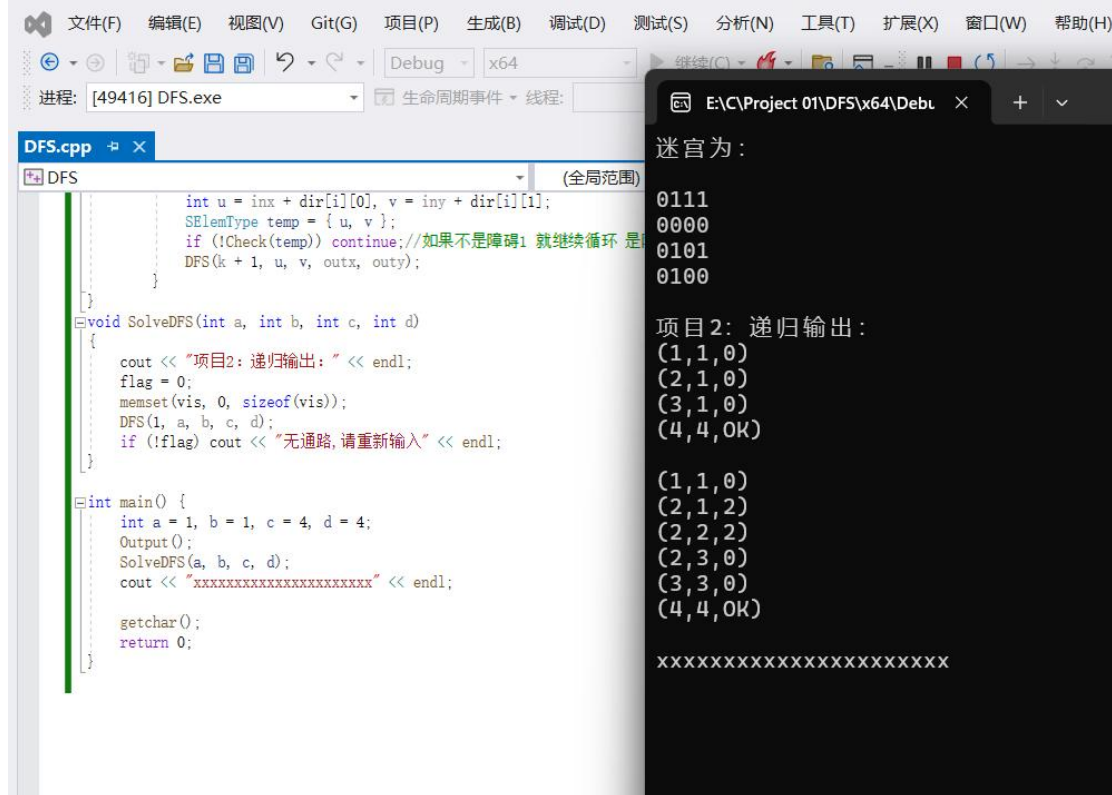
```
5
120
120
```

四、迷宫问题

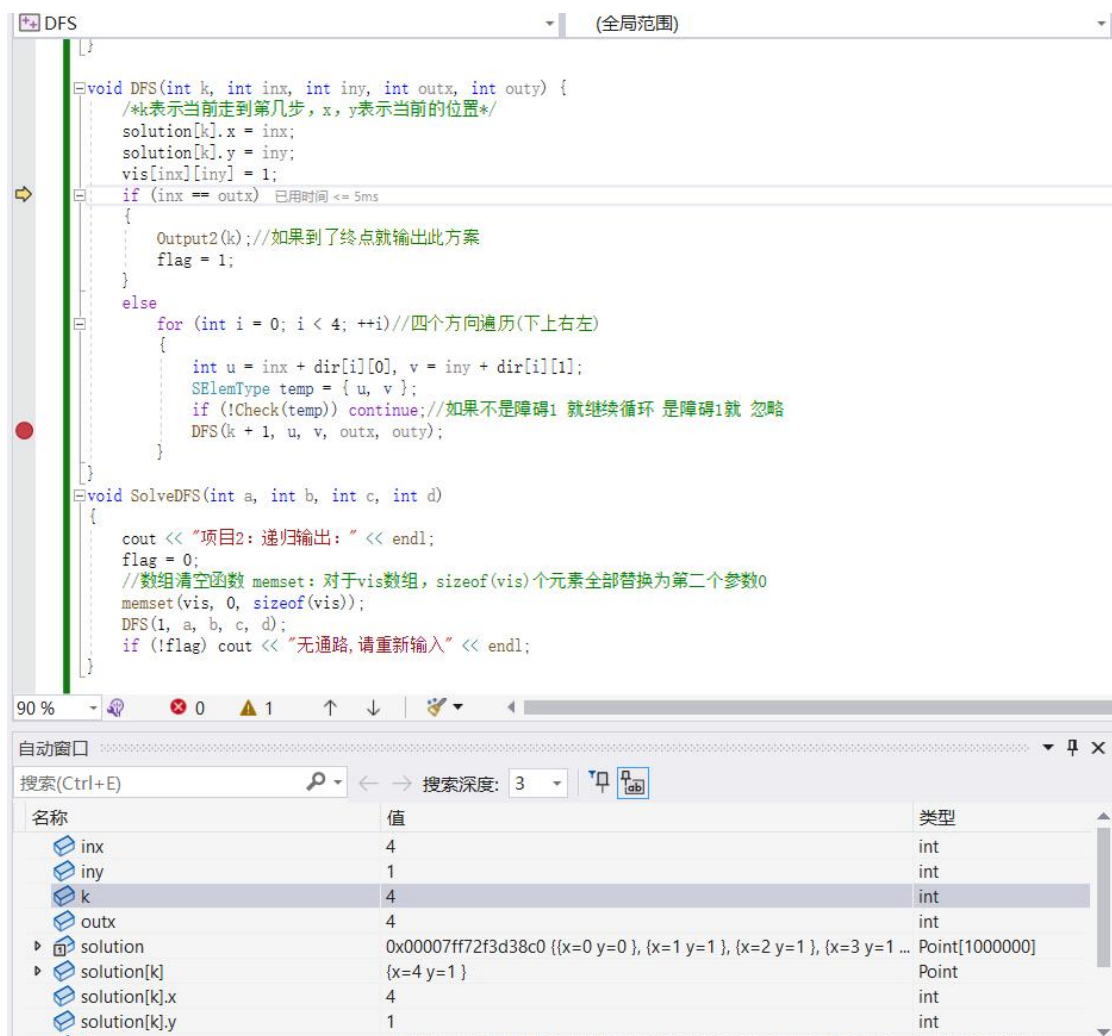
4.1 DFS 递归实现

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

DFS with stack (全局范围) main()

```

        can[i][j] = true;
    }
    if (can[2][1])
    {
        push(&head, { 2, 1, {1} });
        can[2][1] = false;
    }
    if (can[1][2])
    {
        push(&head, { 1, 2, {3} });
        can[1][2] = false;
    }
    can[1][1] = false;
    while (head)
    {
        Point p = pop(&head);
        int p1[36] = { 0 };
        if (can[p.x, p.y + 1])
        {
            append(p.path, 1, p1);
            push(&head, { p.x, p.y + 1, *p1 });
        }
    }

```

自动窗口

搜索深度: 3

名称	值
can	0x00007ff6cc1e0440 {0x00007ff6cc1e0440 {false, false, false, false, ...
can[0]	0x00007ff6cc1e0440 {false, false, false, false, false, false}
can[1]	0x00007ff6cc1e0446 {false, false, false, true, true, false}
can[2]	0x00007ff6cc1e044c {false, false, false, false, false, false}
can[3]	0x00007ff6cc1e0452 {false, true, true, true, true, false}
can[4]	0x00007ff6cc1e0458 {false, false, true, true, true, false}
can[5]	0x00007ff6cc1e045e {false, false, false, false, false, false}
can[1]	0x00007ff6cc1e0446 {false, false, false, true, true, false}
can[1][1]	false
can[1][2]	false
head	0x0000021096ae8c70 {data={x=2 y=1 path=0x0000021096ae8c7...

名称	值	类型
&head	0x00000057524feb58 {0x0000021096aeaca0 {data={x=3 y=1 pat...	Node * *
data	0x0000021096aeaca0 {data={x=3 y=1 path=0x0000021096aeaca...	Node *
next	{x=3 y=1 path=0x0000021096aeaca8 {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	Point
*p1	0x0000021096aea9d0 {data={x=2 y=0 path=0x0000021096aea9d...	Node *
p1	1	int
can	0x00007ff6cc1e0440 {0x00007ff6cc1e0440 {false, false, false, false...	bool[6][6]
can[p.x+1]	0x00007ff6cc1e0452 {false, false, true, true, true, false}	bool[6]
can[p.x+1][p.y]	false	bool
can[p.x-1,p.y]	0x00007ff6cc1e0446 {false, false, false, true, true, false}	bool[6]
p	{x=2 y=1 path=0x00000057524febe8 {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...	Point
p.x	2	int
p.y	1	int

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

The screenshot shows a Visual Studio IDE with two windows. The left window, titled 'DFS with stack.cpp', contains C++ code for a Depth-First Search (DFS) algorithm. The code uses a stack to explore a grid, starting from (1,1) and ending at (4,4). It includes comments in Chinese explaining the steps, such as pushing the start point, checking for the end point, and exploring four directions. The right window, titled 'Microsoft Visual Studio 调试', shows the debug output. It displays a 4x4 grid of 0s and 1s, followed by the path sequence: (1,1,1) -> (2,1,3) -> (2,2,3) -> (2,3,3) -> (2,4,1) -> (3,4,1) -> (4,4). Below the path, a message states: 'E:\C\Project 01\DFS with stack\x64\Debug\DFS with stack.exe (进程 52448) 要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自按任意键关闭此窗口...”。'

```
int main() {
    Node* head = nullptr;
    for (int i = 1; i <= 4; i++) {
        for (int j = 1; j <= 4; j++) {
            int x;
            cin >> x;
            if (x == 0) {
                can[i][j] = true;
            }
        }
    }
    push(&head, { 1, 1, { } }); // 将起点加入栈
    can[1][1] = false;

    while (head) {
        Point current = pop(&head);

        if (current.x == 4 && current.y == 4) { // 到达终点
            output(current.path);
            break; // 找到一条路径就退出
        }

        // 四个方向: 1向下, 2向上, 3向右, 4向左
        for (int dir = 1; dir <= 4; ++dir) {
            int nx = current.x, ny = current.y;
            switch (dir) {
                case 1: nx++; break;
                case 2: nx--; break;
                case 3: ny++; break;
                case 4: ny--; break;
            }
            if (can[nx][ny]) {
                can[nx][ny] = false; // 标记为已访问
                Point newPoint = { nx, ny, current.path };
                newPoint.path.push_back(dir); // 添加方向到
                push(&head, newPoint); // 加入新点到栈中
            }
        }
    }

    // 清理剩余栈元素
    while (head) pop(&head);

    return 0;
}
```

```
0 1 1 0
0 0 0 0
1 0 1 0
1 0 1 0
(1,1,1) -> (2,1,3) -> (2,2,3) -> (2,3,3) -> (2,4,1) -> (3,4,1) -> (4,4)

E:\C\Project 01\DFS with stack\x64\Debug\DFS with stack.exe (进程 52448)
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自
按任意键关闭此窗口...”。
```

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。

```

C/C++ (土/可/比/四)
#include <vector>
#include <iostream>

int main() {
    std::vector<int> vec; // 创建一个空的vector

    // 添加元素
    vec.push_back(10);
    vec.push_back(20);

    // 访问元素
    std::cout << "第一个元素: " << vec[0] << std::endl;

    // 使用迭代器访问元素
    for (auto it = vec.begin(); it != vec.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    // 修改元素
    vec[0] = 5;

    // 删除元素
    vec.pop_back(); // 删除最后一个元素

    // 获取vector的大小
    std::cout << "Vector size: " << vec.size() << std::endl;
    getchar();
    return 0;
}

```

第一个元素: 10
10 20
Vector size: 1

4.3 DFS 输出所有通路

修改 DFS 代码 使得递归能输出所有通路 主要修改为撤销标记

进程: [30140] DFS for all solution.exe

生命周期

DFS for all solution.cpp

DFS for all solution

```

printf("(d,d,OK)\n\n", n, m); //到达终点
}

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
        // 此处不再设置flag为1
    }
    else {
        for (int i = 0; i < 4; ++i) { //四个方向
            int u = inx + dir[i][0], v = iny + dir[i][1];
            SElemType temp = { u, v };
            if (!Check(temp)) continue; //如果不满足条件
            DFS(k + 1, u, v, outx, outy);
        }
    }
    vis[inx][iny] = 0; //撤销当前点的访问标记
}

void SolveDFS(int a, int b, int c, int d)
{
    cout << "项目2: 递归输出:" << endl;
}

```

90 %

未找到相关问题

E:\C\Project 01\DFS for all solution

迷宫为:

```

0111
0000
0001
0100

```

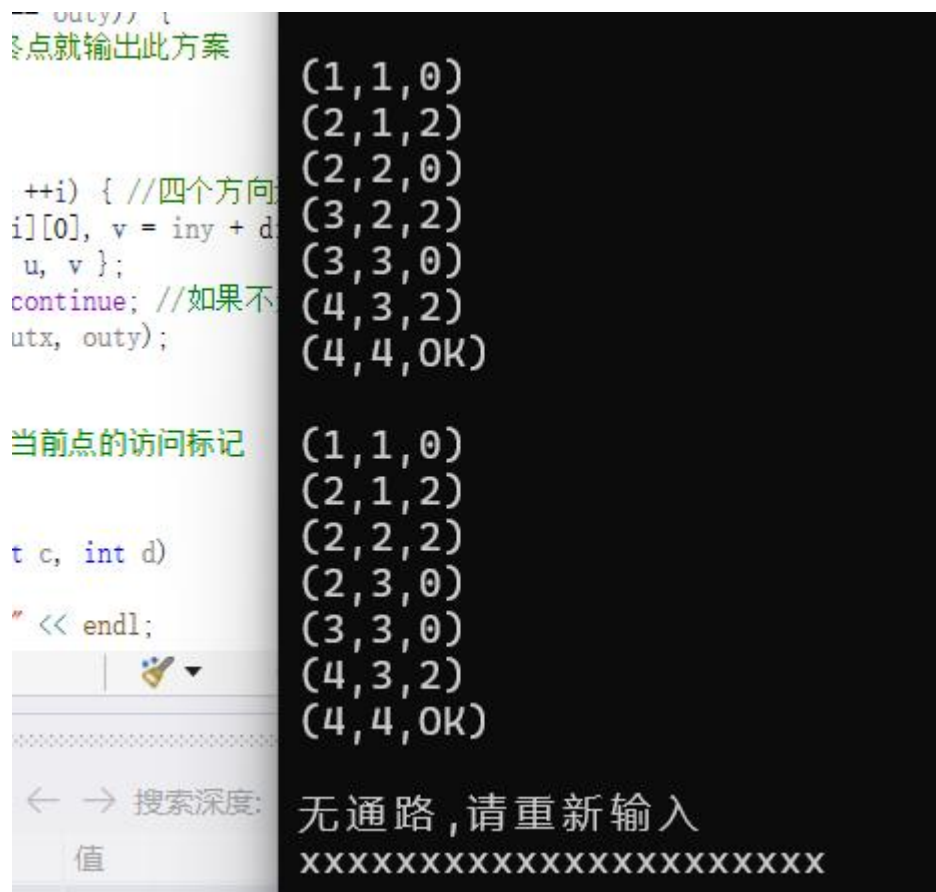
项目2: 递归输出:

```

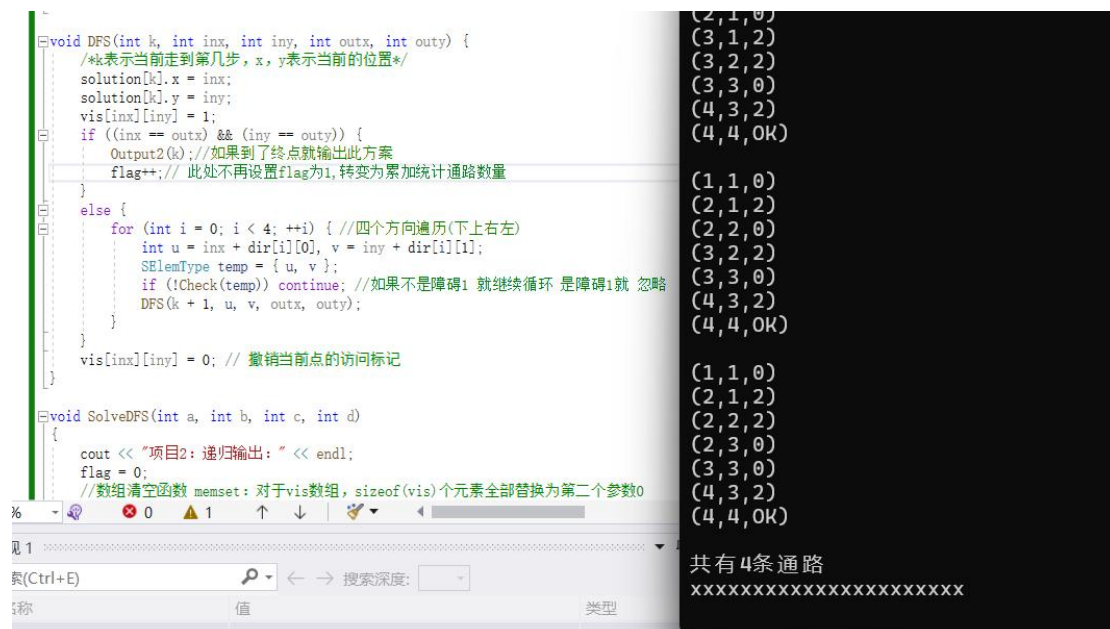
(1,1,0)
(2,1,0)
(3,1,2)
(3,2,1)
(2,2,2)
(2,3,0)
(3,3,0)
(4,3,2)
(4,4,OK)

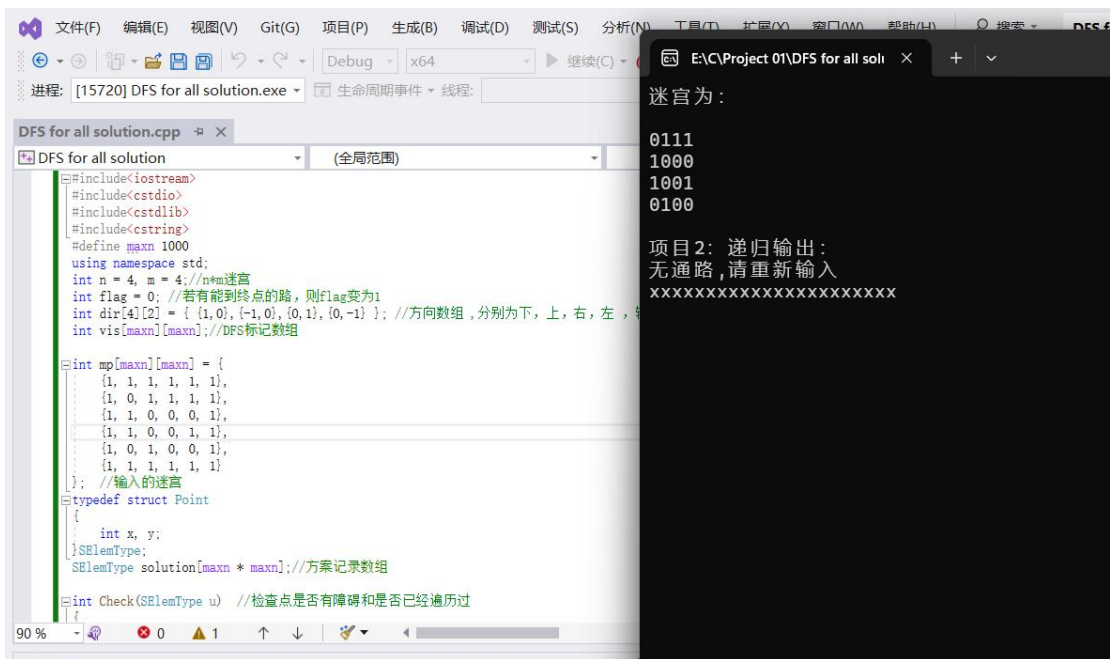
(1,1,0)
(2,1,0)
(3,1,2)
(3,2,2)
(3,3,0)
(4,3,2)
(4,4,OK)

```



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。

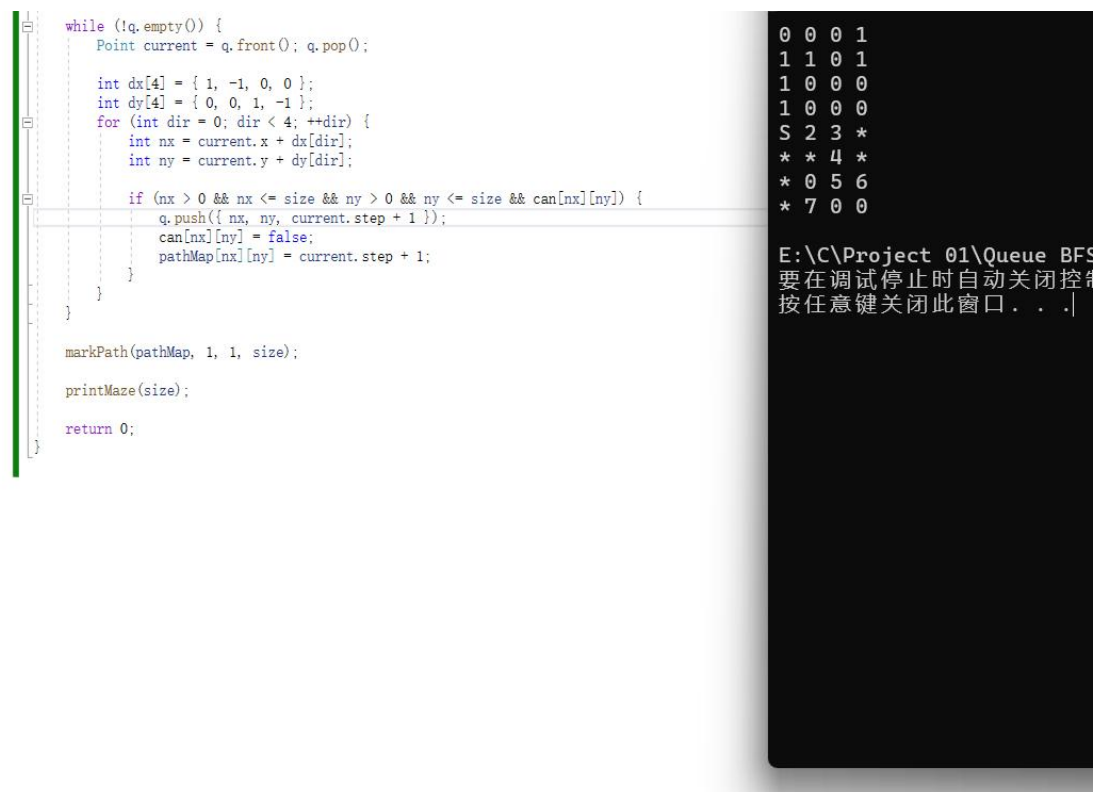
pop(): 移除队首的元素。

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。



```
while (!q.empty()) {
    Point current = q.front(); q.pop();

    int dx[4] = { 1, -1, 0, 0 };
    int dy[4] = { 0, 0, 1, -1 };
    for (int dir = 0; dir < 4; ++dir) {
        int nx = current.x + dx[dir];
        int ny = current.y + dy[dir];

        if (nx > 0 && nx <= size && ny > 0 && ny <= size && can[nx][ny]) {
            q.push({ nx, ny, current.step + 1 });
            can[nx][ny] = false;
            pathMap[nx][ny] = current.step + 1;
        }
    }

    markPath(pathMap, 1, 1, size);

    printMaze(size);

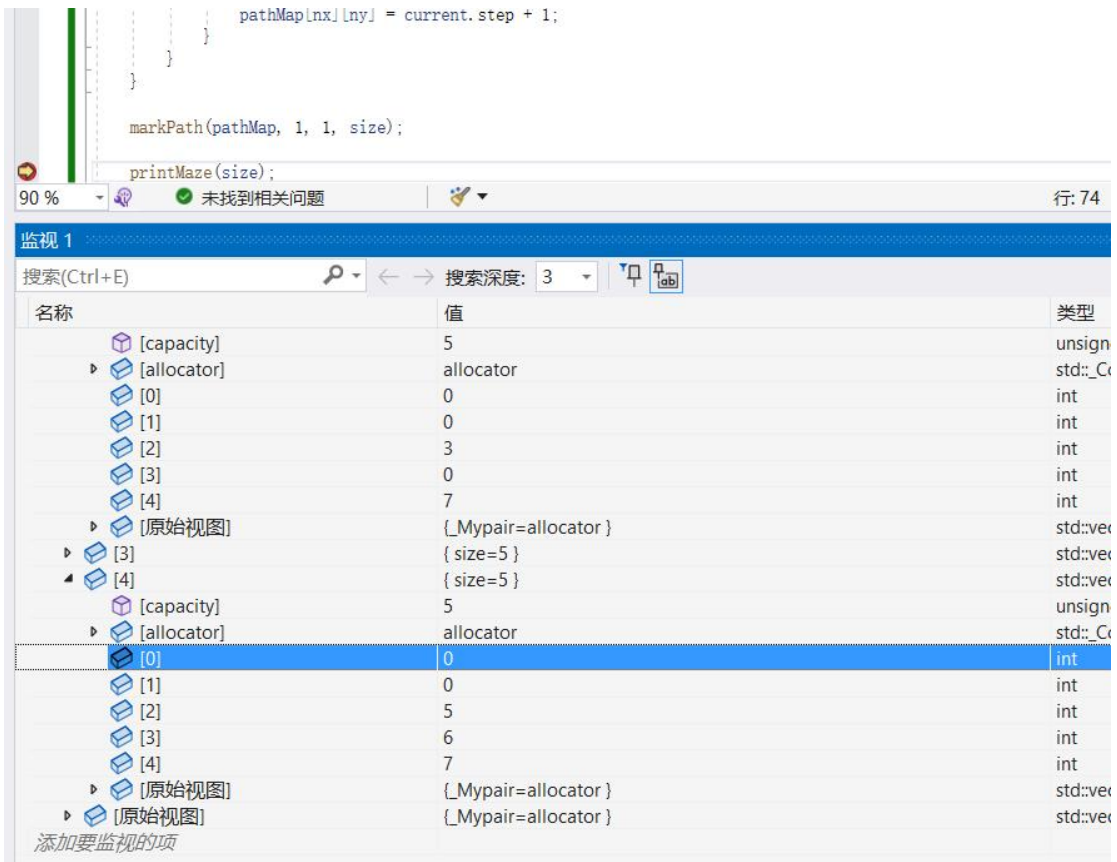
    return 0;
}
```

```
0 0 0 1
1 1 0 1
1 0 0 0
1 0 0 0
S 2 3 *
* * 4 *
* 0 5 6
* 7 0 0

E:\C\Project 01\Queue BFS
要在调试停止时自动关闭控制台
按任意键关闭此窗口...|
```

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.



输入迷宫

0	0	0	1
1	0	1	0
1	0	0	0
1	0	0	0

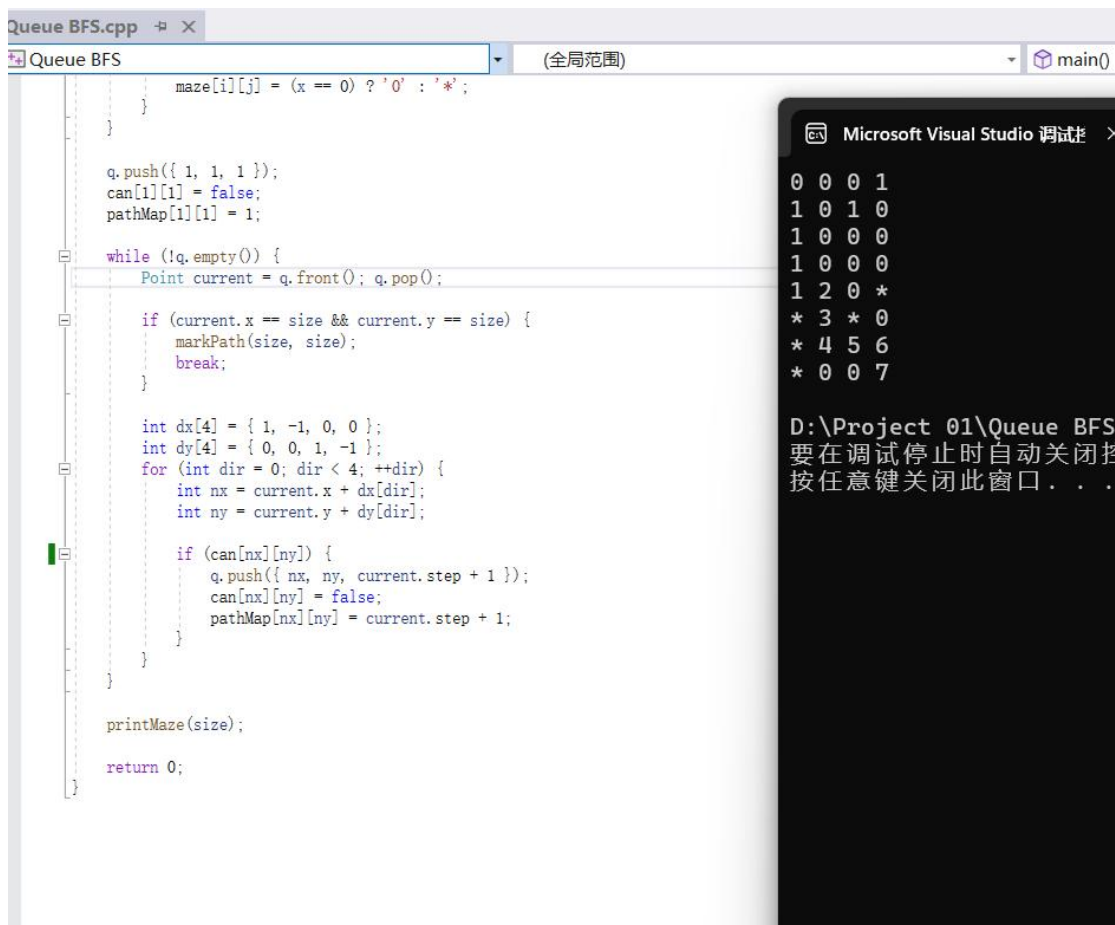
理想 pathMap

0	1	2	*
*	2	*	6
*	3	4	5
*	4	5	6

实际 pathMap

1	2	3	0
0	3	0	7
0	4	5	6
0	5	6	7

所以代码错误在于未能正确判断终点.



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

4.5.1 DFS 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

4.5.2 A* 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 (!isEmpty(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。