**程序设计与算法综合训练**

**实验报告**

|  |  |
| --- | --- |
| **实验名称**： | 迷宫问题的求解 |
| **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](#_Toc184025809)

[1.1 实验目的 4](#_Toc184025810)

[1.2 实验内容 4](#_Toc184025811)

[1.3 实验要求 4](#_Toc184025812)

[1.4 实验任务 5](#_Toc184025813)

[二、 任务1：栈与队列的顺序、链式结构的实现 6](#_Toc184025814)

[2.1 栈的基本概念结构实现 6](#_Toc184025815)

[2.1.1 栈的基本概念 6](#_Toc184025816)

[2.1.2 顺序栈的代码 6](#_Toc184025817)

[2.1.3 顺序栈的调试 7](#_Toc184025818)

[2.1.4 链栈代码 8](#_Toc184025819)

[2.1.5 链栈的调试 9](#_Toc184025820)

[2.2 队列的基本概念与结构实现 10](#_Toc184025821)

[2.2.1 队列的基本概念 10](#_Toc184025822)

[2.2.2 顺序队列代码 11](#_Toc184025823)

[2.2.3 顺序队列的调试 11](#_Toc184025824)

[2.2.4 链式队列代码 13](#_Toc184025825)

[2.2.5 链队列的调试 14](#_Toc184025826)

[三、 任务2：递归与迭代的区别与联系 15](#_Toc184025827)

[3.1 递归函数 15](#_Toc184025828)

[3.2 递归与迭代区别与联系 15](#_Toc184025829)

[四、 任务3：深搜与广搜的区别、优缺点 16](#_Toc184025830)

[4.1 深度优先搜索 16](#_Toc184025831)

[4.2 广度优先搜索 17](#_Toc184025832)

[五、 任务4：迷宫问题的实现 17](#_Toc184025833)

[5.1 使用BFS非递归实现（队列）： 17](#_Toc184025834)

[5.1.1 关键部分代码如下 17](#_Toc184025835)

[5.1.2 运行结果截图 17](#_Toc184025836)

[5.2 DFS链栈操作，非递归，且加入方向数组 18](#_Toc184025837)

[5.2.1 关键代码 18](#_Toc184025838)

[5.2.2 运行结果 20](#_Toc184025839)

[5.3 求迷宫所有可能的通路（栈） 20](#_Toc184025840)

[5.3.1 思路分析 20](#_Toc184025841)

[5.3.2 运行结果截图 20](#_Toc184025842)

[六、 实验总结与常见问题汇总 21](#_Toc184025843)

[6.1 实验总结 21](#_Toc184025844)

[6.2 常见问题汇总 22](#_Toc184025845)

[6.2.1 配置文件参考 22](#_Toc184025846)

[6.2.2 配置文件常见问题汇总 26](#_Toc184025847)

# 实验内容及要求

## 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。

## 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。

## 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。

问题：

1）编程模拟银行的业务活动；

2）一天结束，计算某银行一天中的平均客户逗留时间；

核心：计算某银行一天服务了多少个客户，以及每个客户逗留时间（排队时长+办理业务时长（随机数））

## 实验任务

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。

# 任务1：栈与队列的顺序、链式结构的实现

## 栈的基本概念结构实现

### 栈的基本概念

1. 定义：只允许在一端进行插入或删除的线性表（数组，链表）
2. 栈顶（top）：允许进行插入或删除的一端
3. 栈底（bottom）：与栈顶相对应的一端
4. 特点：后进先出

栈的存储结构：

1. 数组存储结构：利用连续存储区域来存放栈中的所有元素，通常由一个一维数组和记录栈顶元素位置的变量构成
2. 链式存储结构：链栈，栈顶在头部切运算首先的单链表

栈的基本操作：

1. 进栈（push）：栈的插入操作叫做压栈/入栈，如数据在栈顶
2. 出栈（pop）：栈的删除操作叫做出栈，出数据也在也在栈顶

### 顺序栈的代码

//栈的定义

**typedef** **struct** Stack {

**int** data[MAX\_SIZE];     //用一维数组存储数据

**int** top;                //记录栈顶元素位置

} Stack;

//入栈操作

**void** push(Stack\* s, **int** x) {

**if** (s->top == MAX\_SIZE - 1) {

        cout << “栈上溢出”<<endl;

**return**;

    }

    s->data[++s->top] = x;

}

//出栈操作

**int** pop(Stack\* s) {

**if** (s->top == -1) {

        cout<<“栈下溢出”<<endl;

**return** -1;

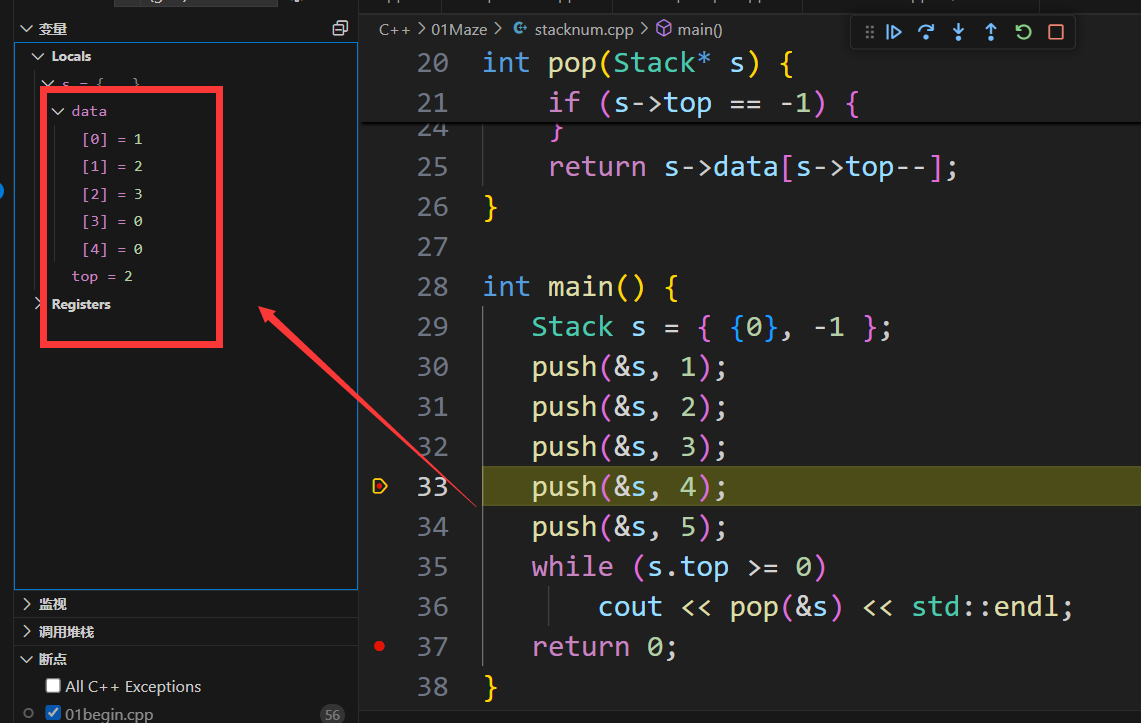
    }

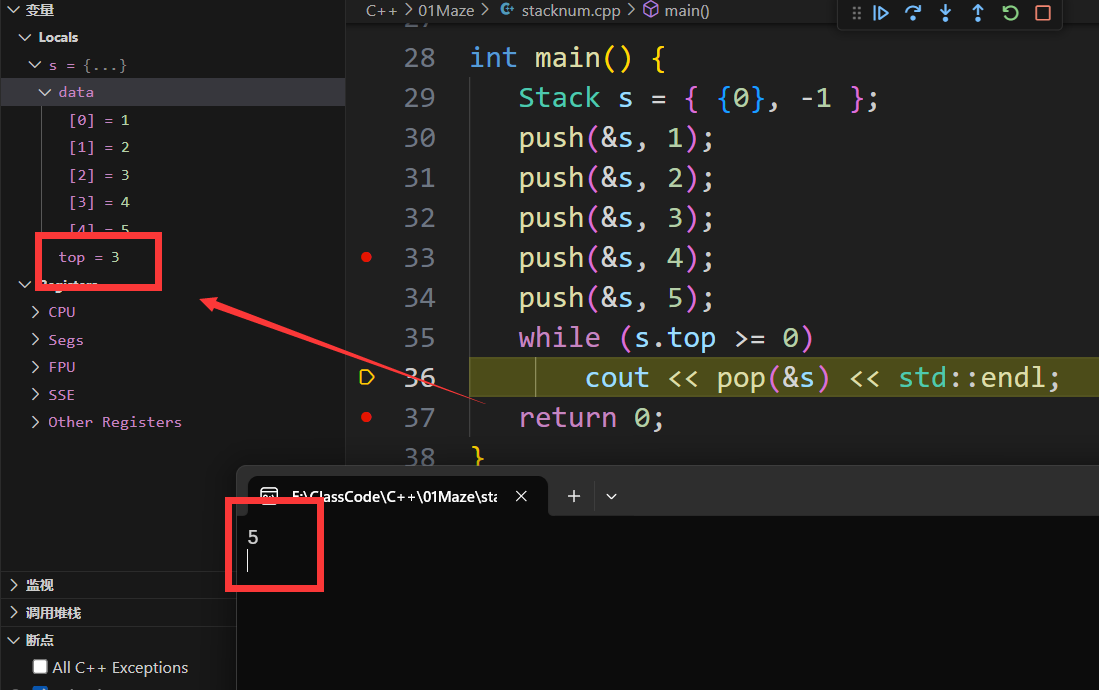
**return** s->data[s->top--];

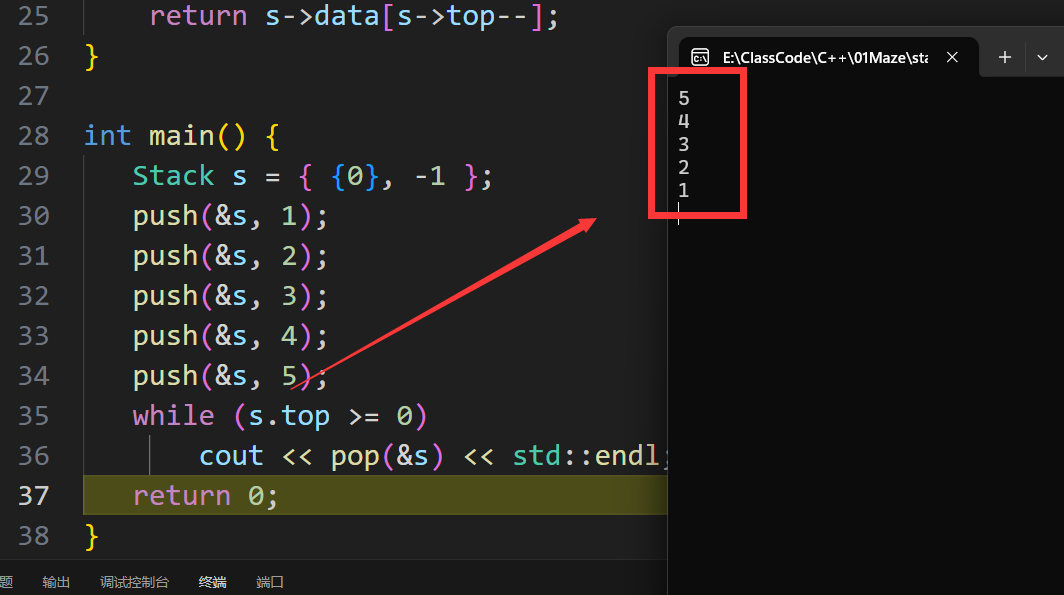
}

### 顺序栈的调试

可以看出栈后进先出的特点







### 链栈代码

typedef struct Node {

int data;

struct Node\* next;

} Node;

void push(Node\*\* head, int data) {

Node\* new\_node = new Node;

new\_node->data = data;

new\_node->next = \*head;

\*head = new\_node;

}

int pop(Node\*\* head) {

if (\*head == NULL) {

cout << "Stack is empty." << endl;

return -1;

}

int data = (\*head)->data;

Node\* temp = \*head;

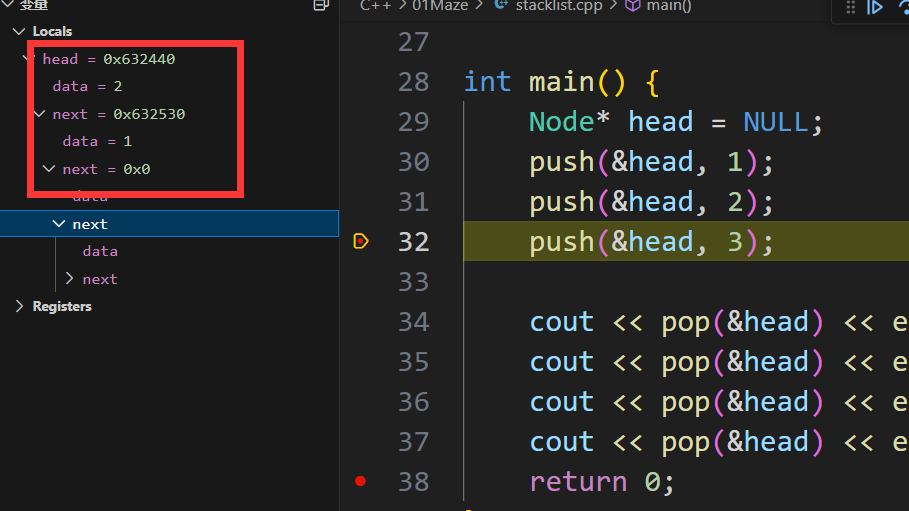
\*head = (\*head)->next;

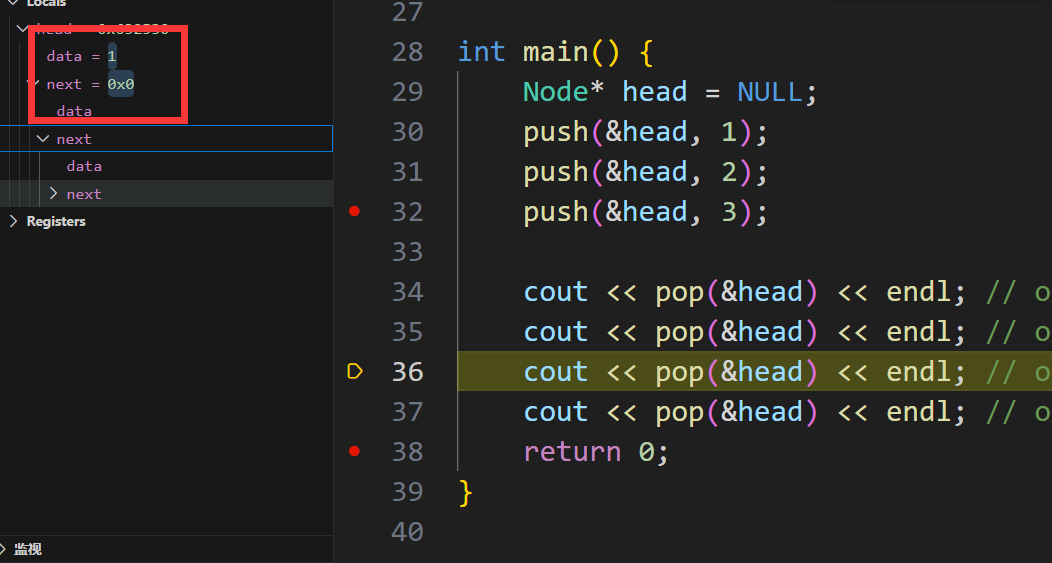
delete temp;

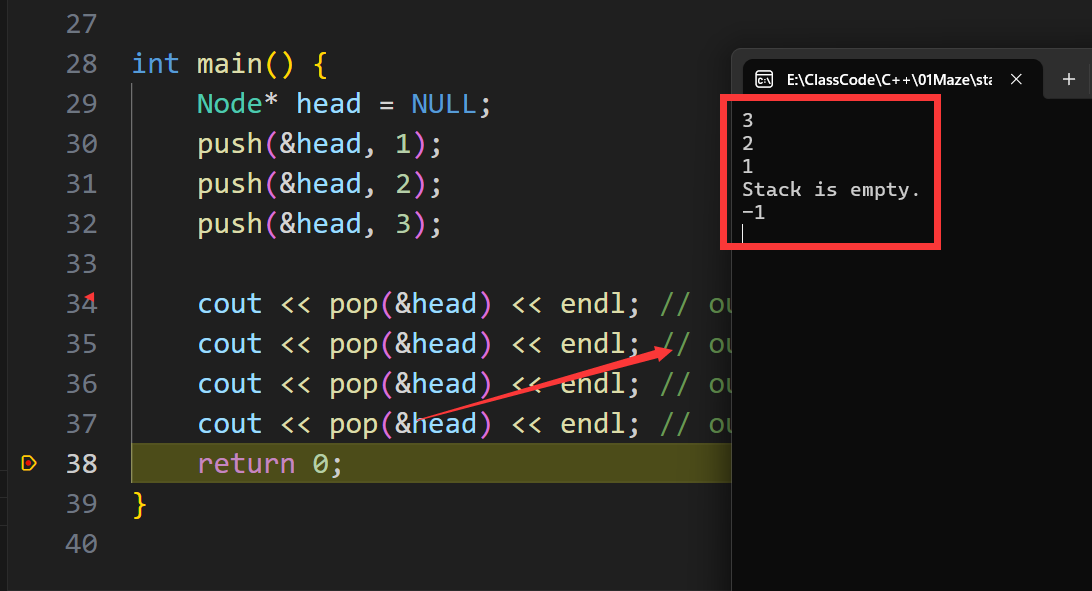
return data;

}

### 链栈的调试







## 队列的基本概念与结构实现

### 队列的基本概念

队列是一种线性数据结构，它具有先进先出（FIFO）的特性。

队列通常有两个基本操作：入队（enqueue）和出队（dequeue）。

入队操作将元素添加到队列的末尾，出队操作则从队列的前端移除元素。

队列可以用数组（顺序队列）或链表（链式队列）来实现。

### 顺序队列代码

//队列结构

**typedef** **struct** {

**int** data[MAX\_SIZE];

**int** front, back;

} Queue;

//入队

**void** enqueue(Queue\* q, **int** x) {

**if** (isFull(q)) {

        printf("Error: Queue is full.\n");

**return**;

    }

    q->data[q-> back ++] = x;

}

//出队

**int** dequeue(Queue\* q) {

**if** (isEmpty(q)) {

        printf("Error: Queue is empty.\n");

**return** -1;

    }

**return** q->data[q->front++];

}

//队首队尾指针

**typedef** **struct** Queue

{

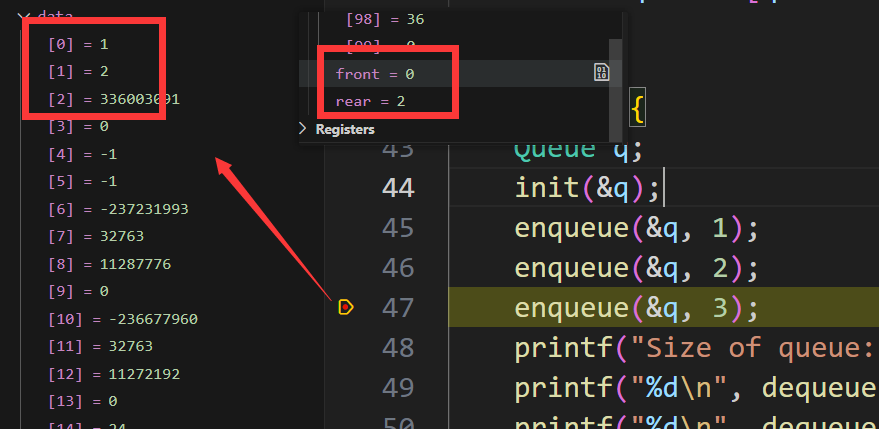
    Node\* head; //队头

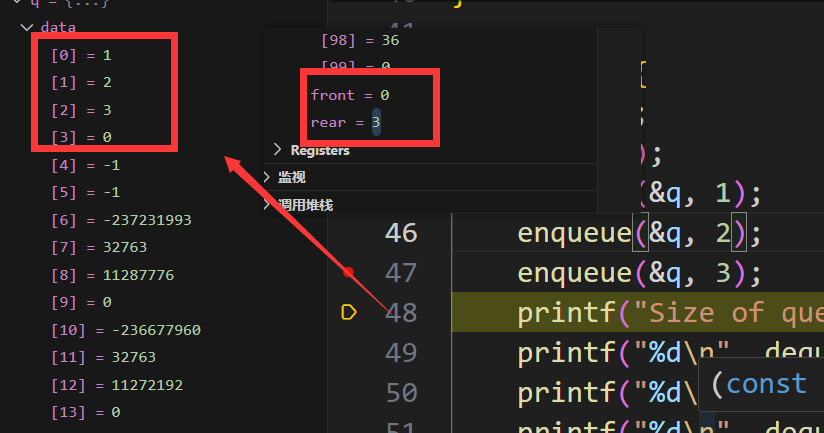
    Node\* tail; //队尾

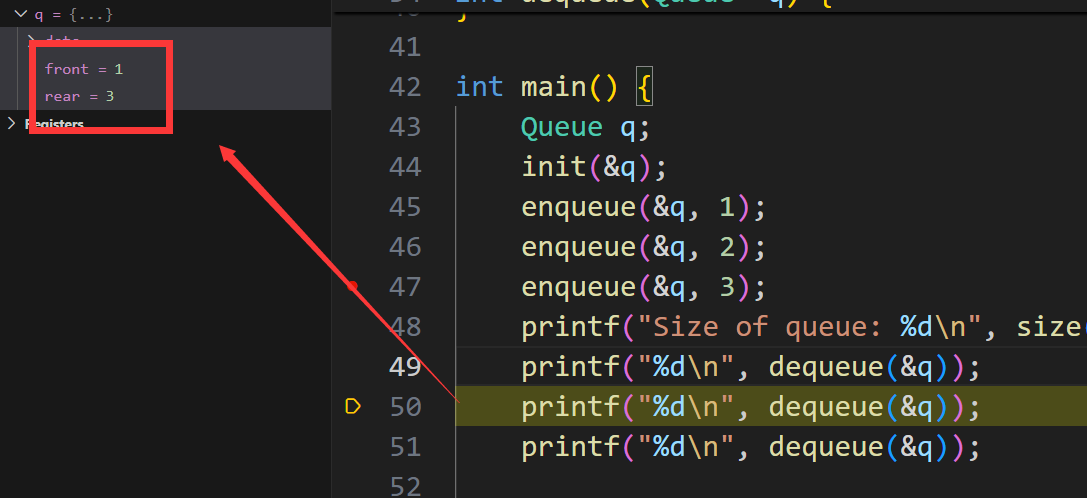
} Queue;

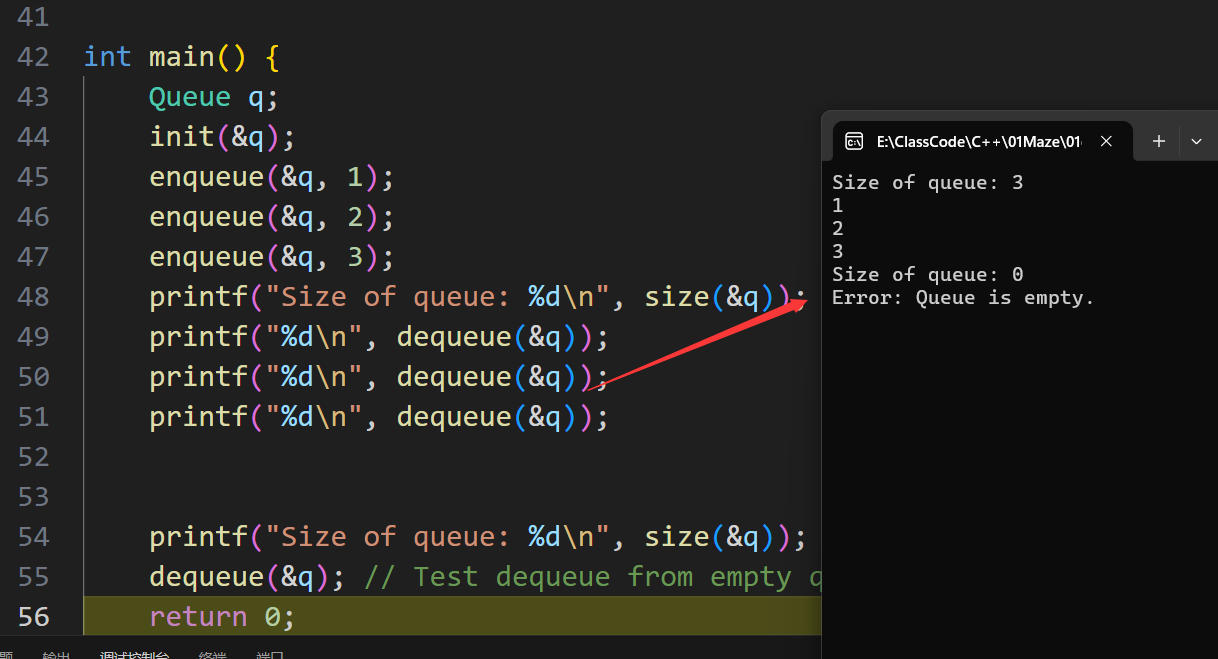
### 顺序队列的调试

可以看出队列先进先出的特点









### 链式队列代码

//队列结点

**typedef** **struct** Node

{

**int** data;

**struct** Node\* next;

} Node;

//队列结点

**void** enqueue(Queue\* q, **int** data)

{

    Node\* new\_node = **new** Node;

    new\_node->data = data;

    new\_node->next = NULL;

**if** (q->head == NULL) //队列为空

        q->head = q->tail = new\_node;

**else** //队列不为空

    {

        q->tail->next = new\_node;

        q->tail = new\_node;

    }

}

//队首队尾指针

**int** dequeue(Queue\* q)

{

**if** (q->head == NULL) //队列为空

    {

        cout << "队列为空" << endl;

**return** -1;

    }

**int** data = q->head->data;

    Node\* temp = q->head;

    q->head = q->head->next;

**if** (q->head == NULL) //队列已经为空

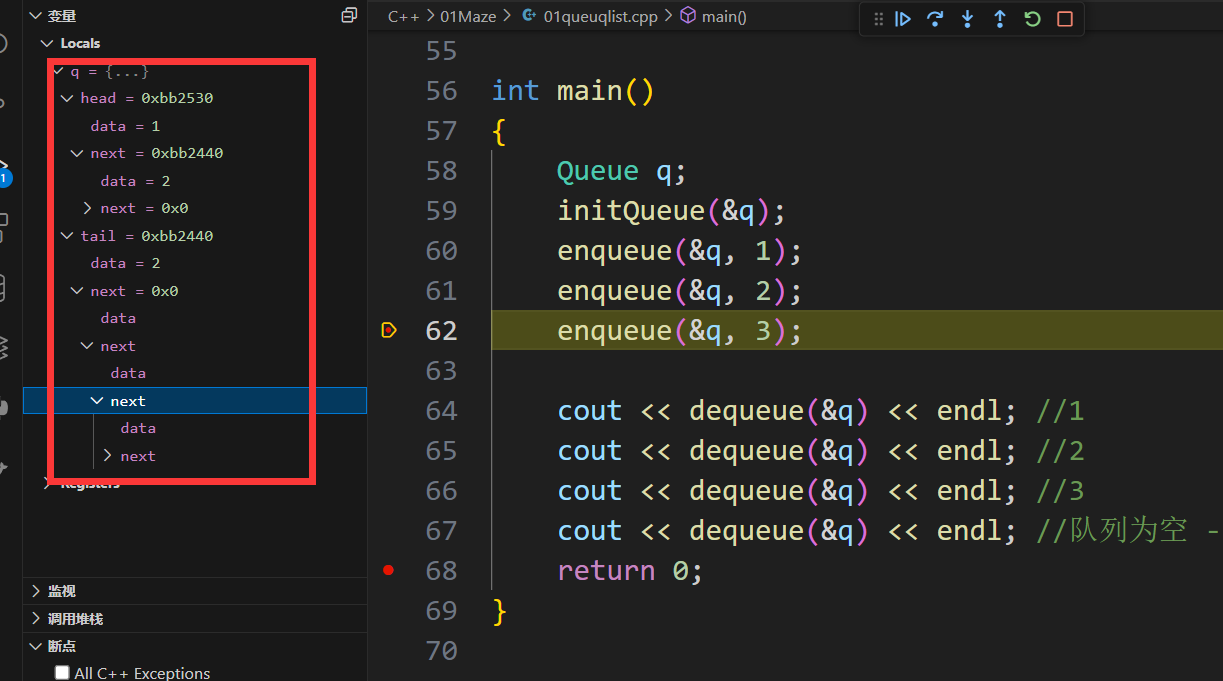
        q->tail = NULL;

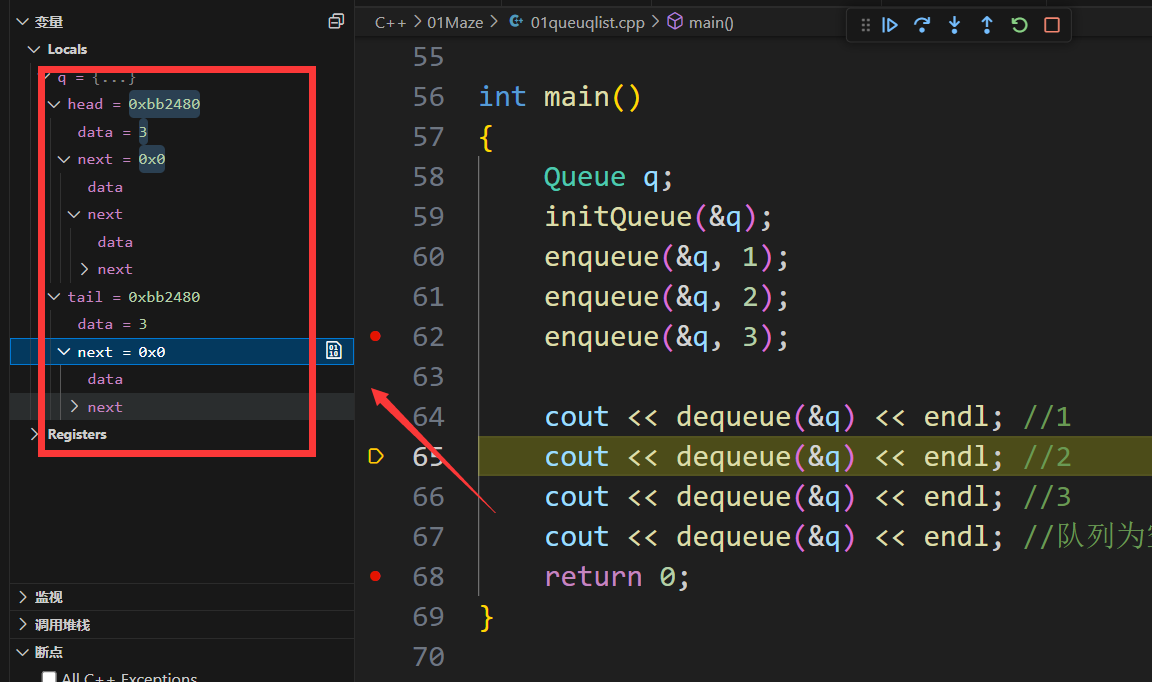
**delete** temp;

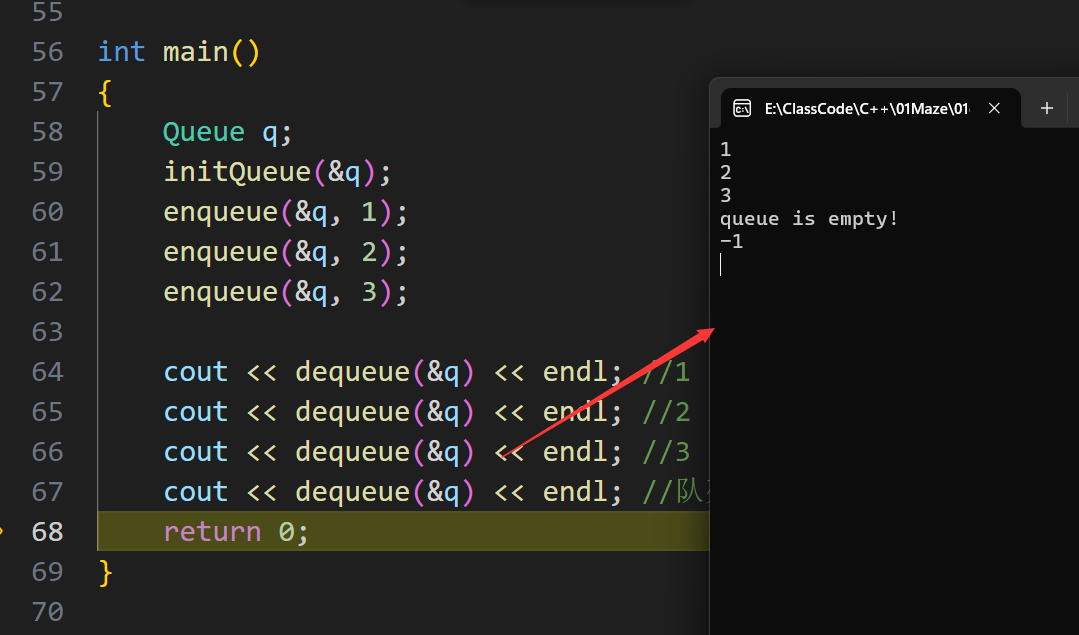
**return** data;

}

### 链队列的调试







# 任务2：递归与迭代的区别与联系

## 递归函数

**递归：**函数直接或间接的调用函数自身

**递归包含两个部分：**

1. 一个基础情况（或终止条件），如在阶乘函数f(x)中，x=1即为终止条件
2. 一个或多个递归情况（或递推条件）

**递归分为两个子过程：**

1. 递过程：函数不断的调用自身，直到走到函数的终止条件，第一阶段结束
2. 归过程：函数不断的返回的过程

## 递归与迭代区别与联系

**递归：**函数子调用

**迭代：**迭代时通过循环的方式重估执行同一段代码来解决问题

1.递归与迭代都是实现循环的方式，但递归是通过函数调用自身实现的，迭代是通过循环实现的。

2. 递归是利用栈来实现函数的调用与返回，可以通过硬件的支持进行优化提高效率。

3.迭代不需要函数的调用和返回操作，也不会存在栈溢出的现象。

# 任务3：深搜与广搜的区别、优缺点

基于树或有向图的两种重要的搜索方法：深度优先搜索，广度优先搜索

## 深度优先搜索

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。

## 广度优先搜索

广度优先搜索算法是一种用于图和树的遍历\搜索算法：

1.广度优先搜索的过程中，算法会维护一个队列来存储遍历的结点，遍历过的结点会被标记为已访问，以避免重复遍历。

2.广度优先搜索常用于查找两个结点之间的最短路径。

### DFS与BFS各自优缺点

深度优先搜索（DFS）和广度优先搜索（BFS）是常用于迷宫问题的两种搜索算法。它们的核心区别在于搜索的策略不同，导致它们在迷宫求解中的表现各异。

1. 深度优先搜索（DFS）DFS优先选择一个方向尽可能深入探索，直到到达死路或找到出口为止。在遇到死路时，回溯到上一个分叉点继续探索其他方向。

优点：内存占用少：DFS仅需维护路径栈，因此所需的内存较少，适合空间有限的环境；实现简单：DFS的实现较为简单，只需使用递归或栈即可。

缺点：可能找不到最短路径 ：DFS有可能首先探索到一个较长的路径而忽视更短的路径，因此不适合要求最短路径的情况；可能陷入死循环 ：若迷宫中存在环路且未设置访问标记，DFS可能会在环中反复循环。

2. 广度优先搜索（BFS）

策略：BFS从起点开始，一层一层向外扩展，首先探索离起点最近的节点，确保在找到第一个解时即为最短路径。

优点： 保证找到最短路径 ：由于BFS按层逐层探索，它保证在找到目标时路径最短，因此适合最短路径问题；不易陷入死循环 ：一般在每个节点被访问时进行标记，减少了反复访问的可能性。

缺点： 内存占用大 ：BFS需要维护队列，且要记录每层所有节点，内存消耗较大，尤其在搜索空间较大的迷宫中；较慢 ：在一些复杂迷宫中，BFS会遍历较多节点，因此在速度上不如DFS。

# 任务4：迷宫问题的实现

## 使用BFS非递归实现（队列）：

### 关键部分代码如下

BFS非递归实现（队列）：在遍历四个方向时，nx，ny不符合规定（超过行的总数/列的总数/小于0/迷宫不能走/已经方位过）时，都要continue，即换方向，

**for** (**int** i = 0; i < 4; i++)

{

**int** nx = cur.x + adj[i].x;

**int** ny = cur.y + adj[i].y;

**if** (nx < 0 || nx >= ROW || ny < 0 || ny >= COL || maze[nx][ny] == 1 ||visited[nx][ny] == VISITED) {

**continue**;

}

**struct** Node temp;

temp.x = nx;

temp.y = ny;

q.push(temp);

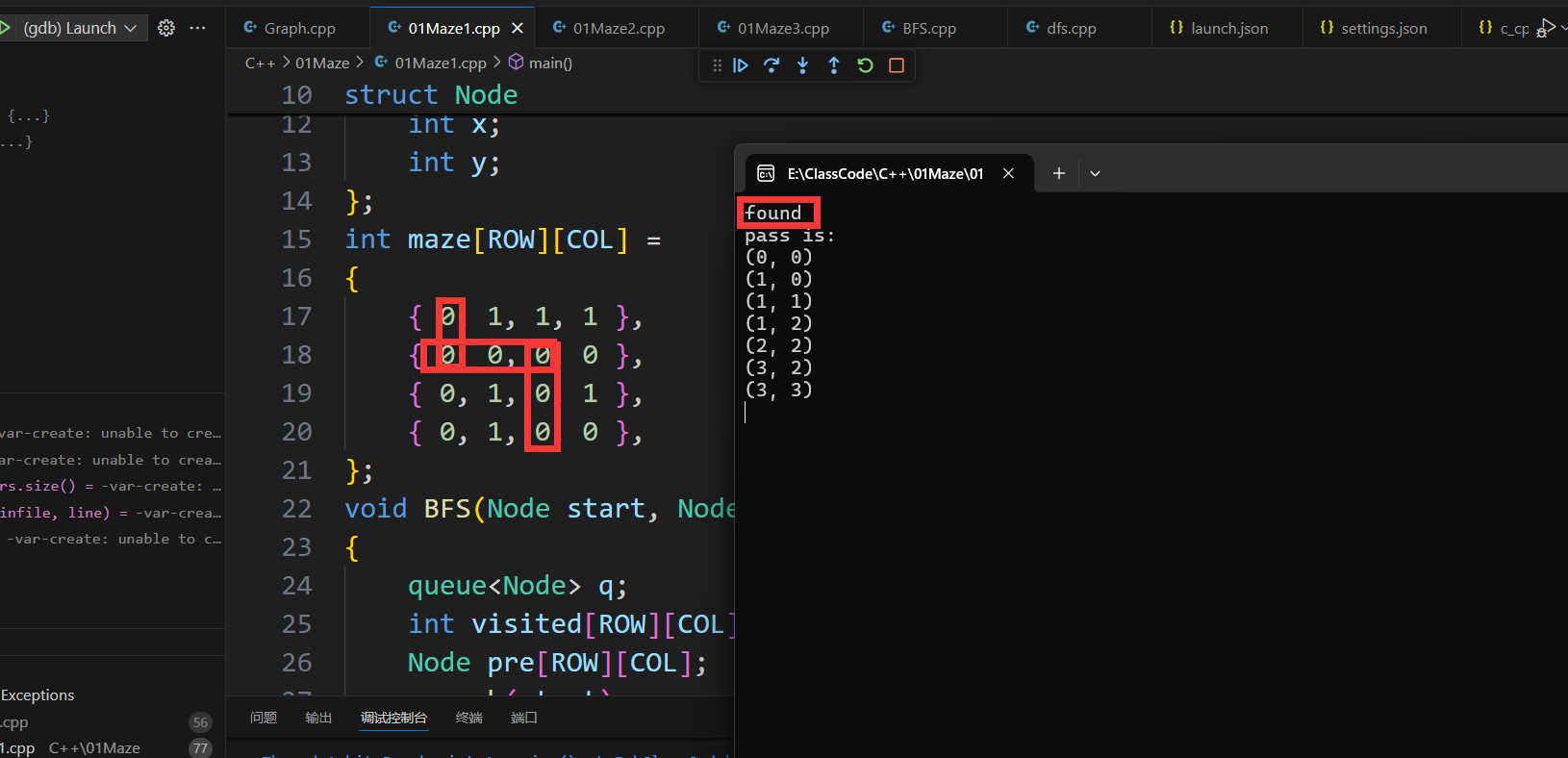
visited[nx][ny] = VISITED;

pre[nx][ny] = cur;

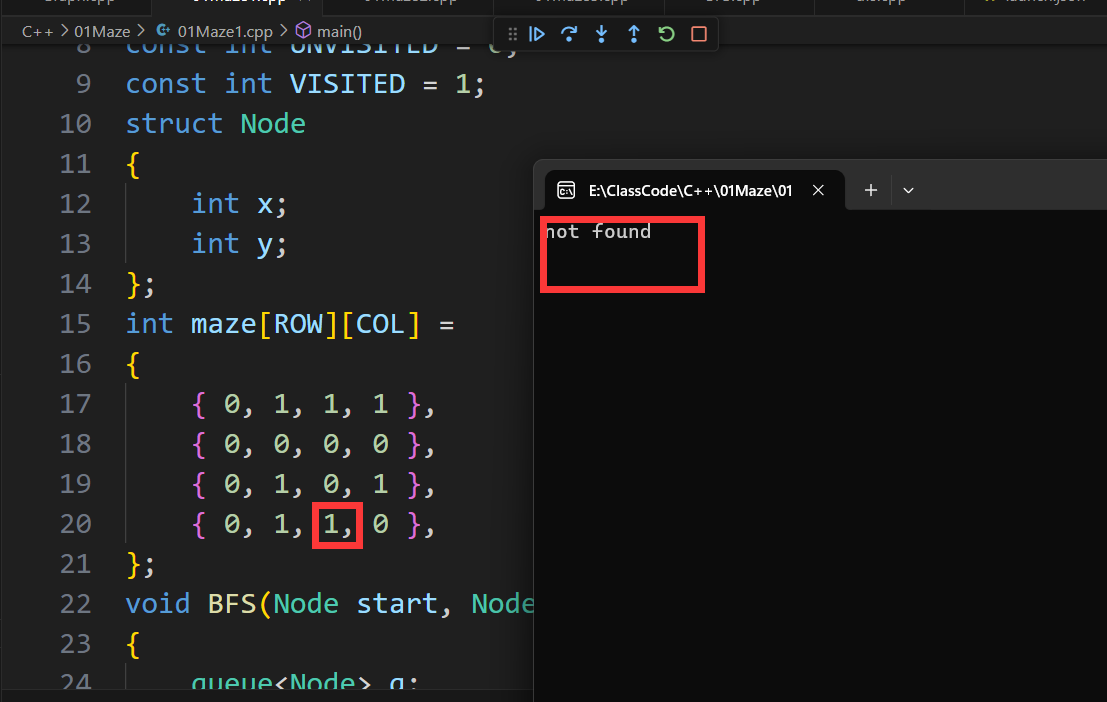
}

### 运行结果截图

当有路径时：



更改迷宫，正确显示无路径



## DFS链栈操作，非递归，且加入方向数组

最后输出时，用#表示迷宫的墙壁，路径上的标号表示方向

### 关键代码

使用memset可以把方向数组全部定义为-1，方便对方向的重新赋值

memset(direct, -1, sizeof(direct));

和前一种方法处理不同的是，不再直接定义二维方向数组，而是分别定义x与y的偏移量，对应标号的x与y的组合可义实现下右上左四个方向，这是为了在输出方向数组时，可以方便的用i的值来表示direct

int dx[4] = {1, 0, -1, 0};

int dy[4] = {0, 1, 0, -1};//分别为下右上左,方向数组用0，1，2，3表示

循环部分代码：

**for**(i = 0; i < 4; i++){

**int** nx = S->next->data.x + dx[i];

**int** ny = S->next->data.y + dy[i];

**if**(nx>=0&&ny>=0&&nx<ROW&&ny<COL&&maze[nx][ny]==0&&visited[nx][ny]==UNVISITED){

                direct[S->next->data.x][S->next->data.y] = i;

                visited[nx][ny] = VISITED;

                SElemType data = {nx, ny};

                Push(S, data);

**break**;

            }

输出#和方向数组的代码如下：

**for**(**int** i = 0; i < ROW; i++){

**for**(**int** j = 0; j < COL; j++){

**if**(maze[i][j] == 1){

                        cout<<setw(4)<<"#";

                    }

**else**{

                        cout<<setw(4)<<direct[i][j];

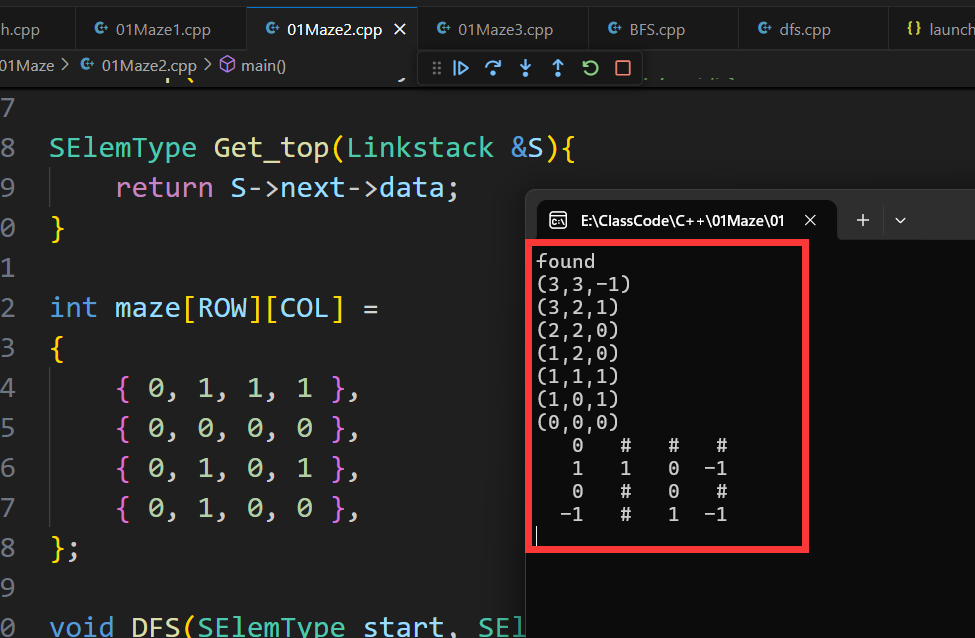
                    }

                }

                cout<<endl;

            }

### 运行结果



迷宫无解的情况与任务一相同，不再展示

## 求迷宫所有可能的通路（栈）

### 思路分析

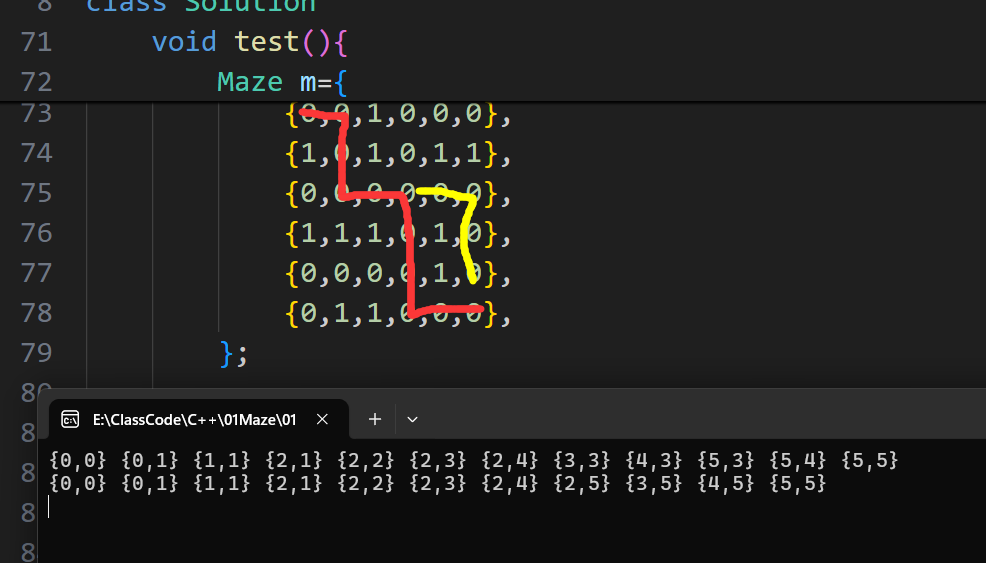
在本任务中，对于每一个可行的节点压入栈中，对于栈中每一个节点，寻找其可以访问的邻居节点压入队列，找到一条通路后，就把路径压入一个定义类型为路径的栈中。

### 运行结果截图

一条通路的情况



当迷宫中有多条通路时，可以分别输出：



# 实验总结与常见问题汇总

## 实验总结

这次实验加深了我对栈与队列的理解，尤其是在顺序结构和链式结构中的应用。通过实现这两种结构，我掌握了它们的基本操作，并在VSCode调试代码时提高了排查错误的效率，对调试工具有了更深入的了解。递归与迭代的比较让我更清楚地认识到递归的简洁性和潜在的性能问题，而迭代方法在大数据处理上更具优势。学习广度优先搜索（BFS）和深度优先搜索（DFS）让我理解了这两种算法的核心思想及其优缺点，帮助我在实际应用中选择合适的算法。通过迷宫求解的非递归与递归实现，我将这些理论应用到实践中，进一步提升了我的编程与算法设计能力。常见问题汇总

## 配置文件参考

Vscode运行C++的配置文件：

{

    "configurations": [

      {

        "name": "Win64",

        "includePath": ["${workspaceFolder}/\*\*"],

        "defines": ["\_DEBUG", "UNICODE", "\_UNICODE"],

        "windowsSdkVersion": "10.0.18362.0",

        // "compilerPath": "D:/January/APP/MinGW/bin/g++.exe",

        "compilerPath": "D:/January/APP/MinGW/bin/gcc.exe",

        "cStandard": "c17",

        "cppStandard": "c++17",

        "intelliSenseMode": "gcc-x64",

        "compilerArgs": [

          "-fexec-charset=GBK"

        ]

      }

    ],

    "version": 4

  }

Launch.json

{

    "version": "0.2.0",

    "configurations": [

      {

        "name": "(gdb) Launch",

        "type": "cppdbg",

        "request": "launch",

        // "program": "${fileDirname}\\${fileBasenameNoExtension}.exe",

        "program": "${fileDirname}\\${fileBasenameNoExtension}.exe",

        "args": [],

        "stopAtEntry": false,

        "cwd": "${workspaceRoot}",

        "environment": [],

        "externalConsole": true,

        "MIMode": "gdb",

        "miDebuggerPath": "D:\\January\\APP\\MinGW\\bin\\gdb.exe",

        "preLaunchTask": "mytask",

        // "preLaunchTask": "g++",

        "setupCommands": [

          {

            "description": "Enable pretty-printing for gdb",

            "text": "-enable-pretty-printing",

            "ignoreFailures": true

          }

        ]

      }

    ]

  }

setting.json

{

    "files.associations": {

      "\*.py": "python",

      "iostream": "cpp",

      "\*.tcc": "cpp",

      "string": "cpp",

      "unordered\_map": "cpp",

      "vector": "cpp",

      "ostream": "cpp",

      "new": "cpp",

      "typeinfo": "cpp",

      "deque": "cpp",

      "initializer\_list": "cpp",

      "iosfwd": "cpp",

      "fstream": "cpp",

      "sstream": "cpp",

      "map": "c",

      "stdio.h": "c",

      "algorithm": "cpp",

      "atomic": "cpp",

      "bit": "cpp",

      "cctype": "cpp",

      "clocale": "cpp",

      "cmath": "cpp",

      "compare": "cpp",

      "concepts": "cpp",

      "cstddef": "cpp",

      "cstdint": "cpp",

      "cstdio": "cpp",

      "cstdlib": "cpp",

      "cstring": "cpp",

      "ctime": "cpp",

      "cwchar": "cpp",

      "exception": "cpp",

      "ios": "cpp",

      "istream": "cpp",

      "iterator": "cpp",

      "limits": "cpp",

      "memory": "cpp",

      "random": "cpp",

      "set": "cpp",

      "stack": "cpp",

      "stdexcept": "cpp",

      "streambuf": "cpp",

      "system\_error": "cpp",

      "tuple": "cpp",

      "type\_traits": "cpp",

      "utility": "cpp",

      "xfacet": "cpp",

      "xiosbase": "cpp",

      "xlocale": "cpp",

      "xlocinfo": "cpp",

      "xlocnum": "cpp",

      "xmemory": "cpp",

      "xstddef": "cpp",

      "xstring": "cpp",

      "xtr1common": "cpp",

      "xtree": "cpp",

      "xutility": "cpp",

      "stdlib.h": "c",

      "string.h": "c"

    },

    "editor.suggest.snippetsPreventQuickSuggestions": false,

    "aiXcoder.showTrayIcon": true,

    "code-runner.runInTerminal": true,

  }

Task.json

{

    "version": "2.0.0",

    "tasks": [

      {

        "label": "mytask",

        "command": "g++",

        "args": [

          "-g",

          "${file}",

          "-o",

          // "-fexec-charset=GBK",

          "${fileDirname}/${fileBasenameNoExtension}.exe"

        ],

        "problemMatcher": {

          "owner": "cpp",

          "fileLocation": [

            "relative",

            "${workspaceRoot}"

          ],

          "pattern": {

            "regexp": "^(.\*):(\\d+):(\\d+):\\s+(warning|error):\\s+(.\*)$",

            "file": 1,

            "line": 2,

            "column": 3,

            "severity": 4,

            "message": 5

          }

        },

        "group": {

          "kind": "build",

          "isDefault": true

        }

      },

      {

        "type": "cppbuild",

        "label": "C/C++: g++.exe 生成活动文件",

        "command": "D:/January/APP/MinGW/bin/g++.exe",

        "args": [

          "-fdiagnostics-color=always",

          "-g",

          "${file}",

          "-o",

          "${fileDirname}\\${fileBasenameNoExtension}.exe"

        ],

        "options": {

          "cwd": "D:/January/APP/MinGW/bin"

        },

        "problemMatcher": [

          "$gcc"

        ],

        "group": "build",

        "detail": "编译器: D:/January/APP/MinGW/bin/g++.exe"

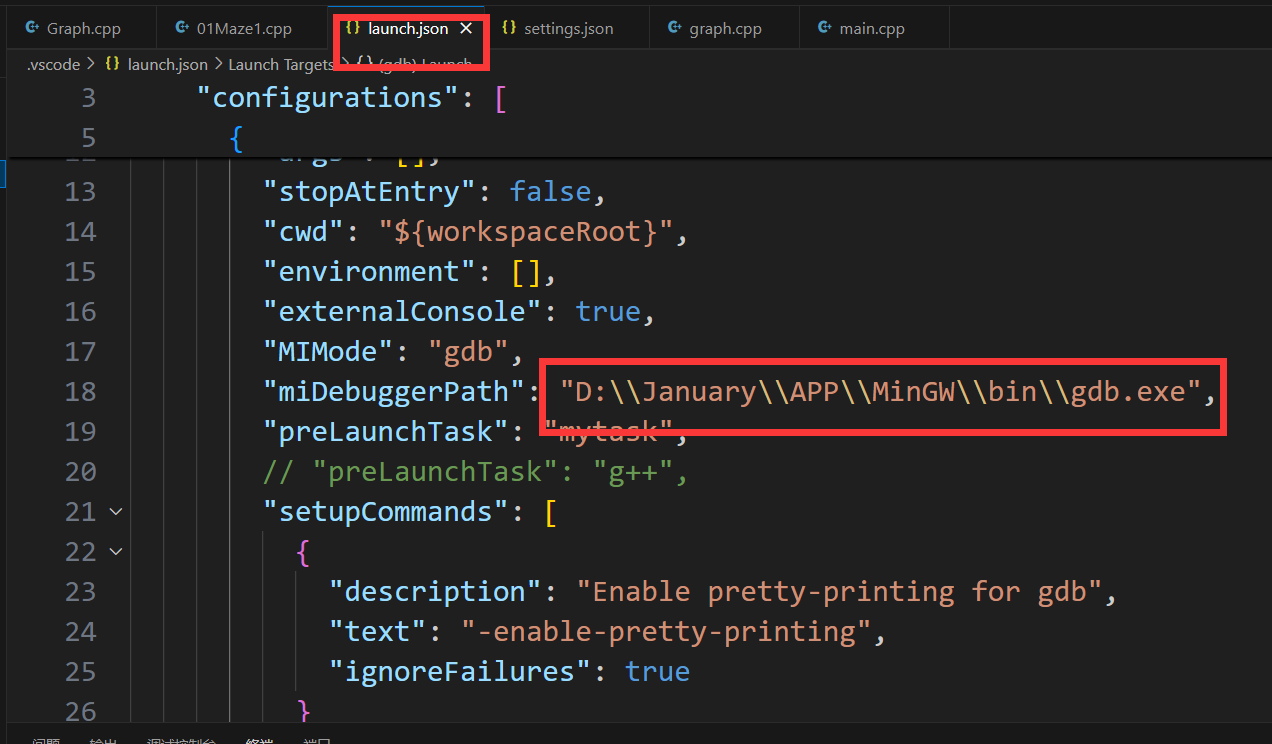
      }

    ]

  }

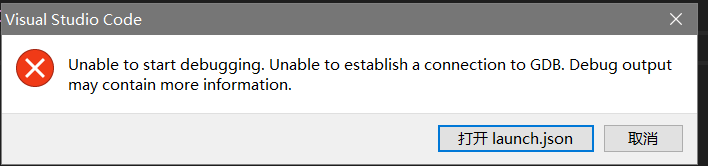
## 配置文件常见问题汇总

用vscode配置c++文件时，可能会出现的问题：



按照教程创建好.vscode文件夹，在launch.json中，这里必须改成自己gdb.exe对应的路径，如果不熟悉相对路径的表示的话，建议直接改成绝对路径

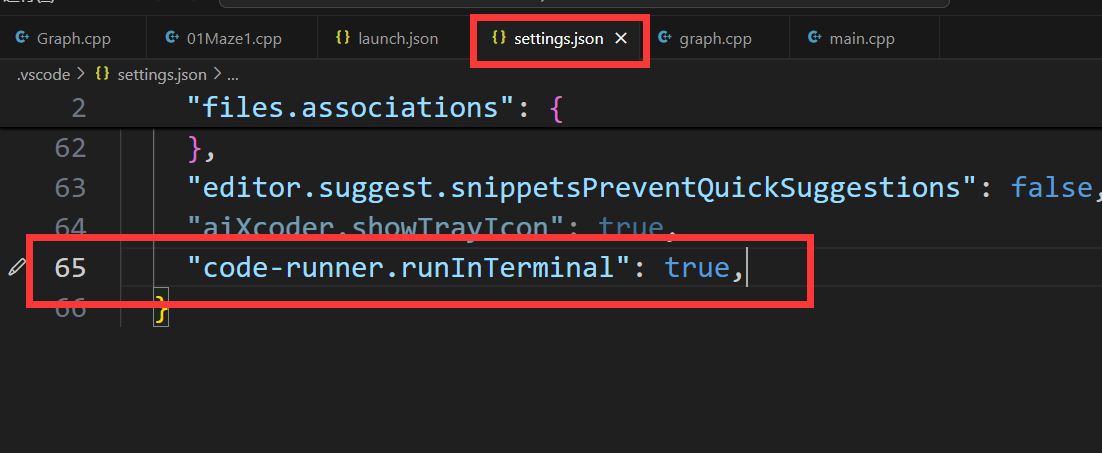
这里也是非常经典的问题，具体打开launch.json，找教程配置就行了



Vscode在调试过程中出现，当需要输入内容时，输出端那里不能输入cin

解决方案：在setting.json中加入以下一段话，就会自动匹配到终端，就可以正常输出

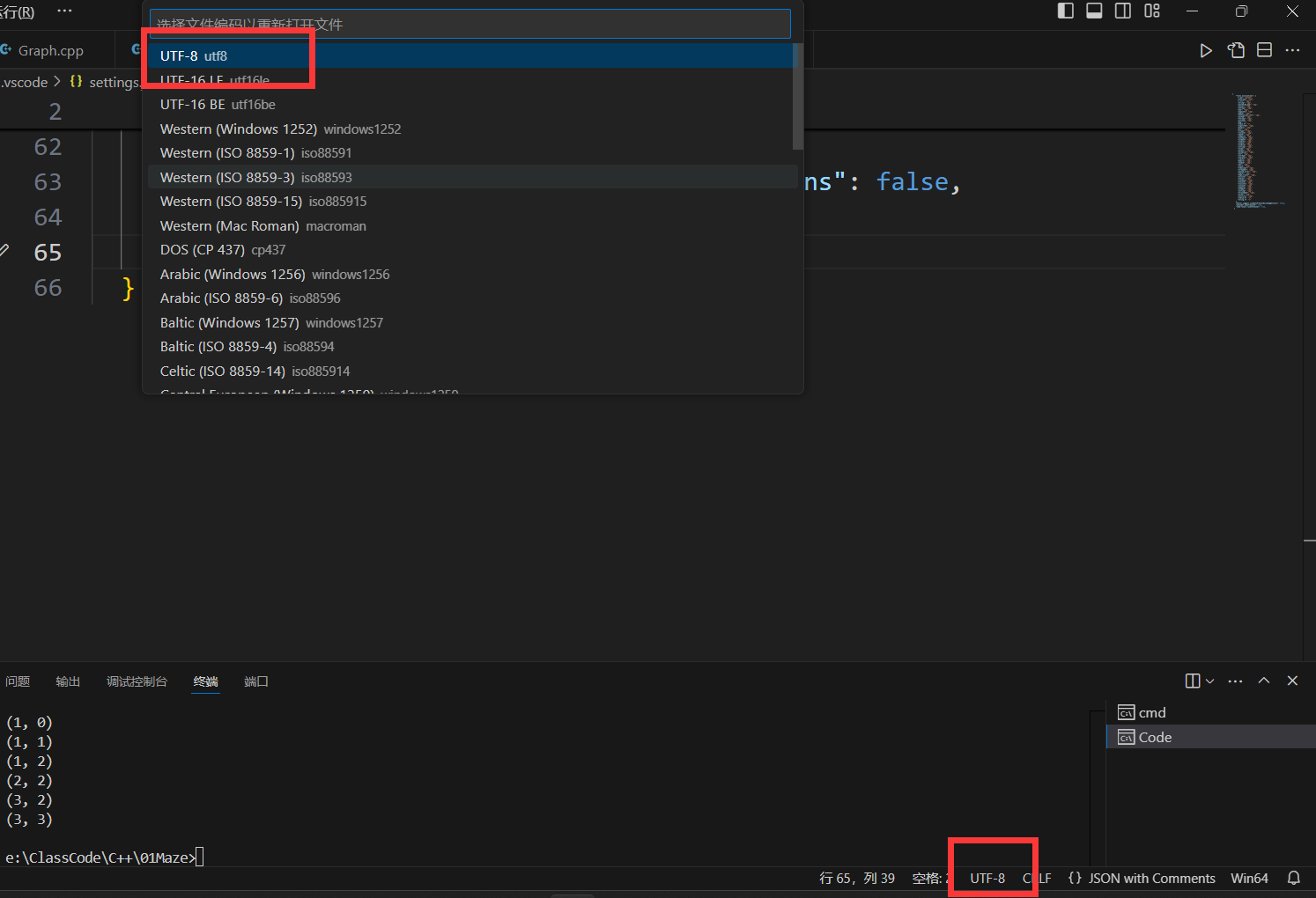
"code-runner.runInTerminal": true,



这样在需要输入时，就会自动匹配到终端

Vscode经常出现中文乱码的情况，分以下几种类型：

如果代码的中文注释是乱码，重新选择编码方式就可以解决，utf-8或者GBK

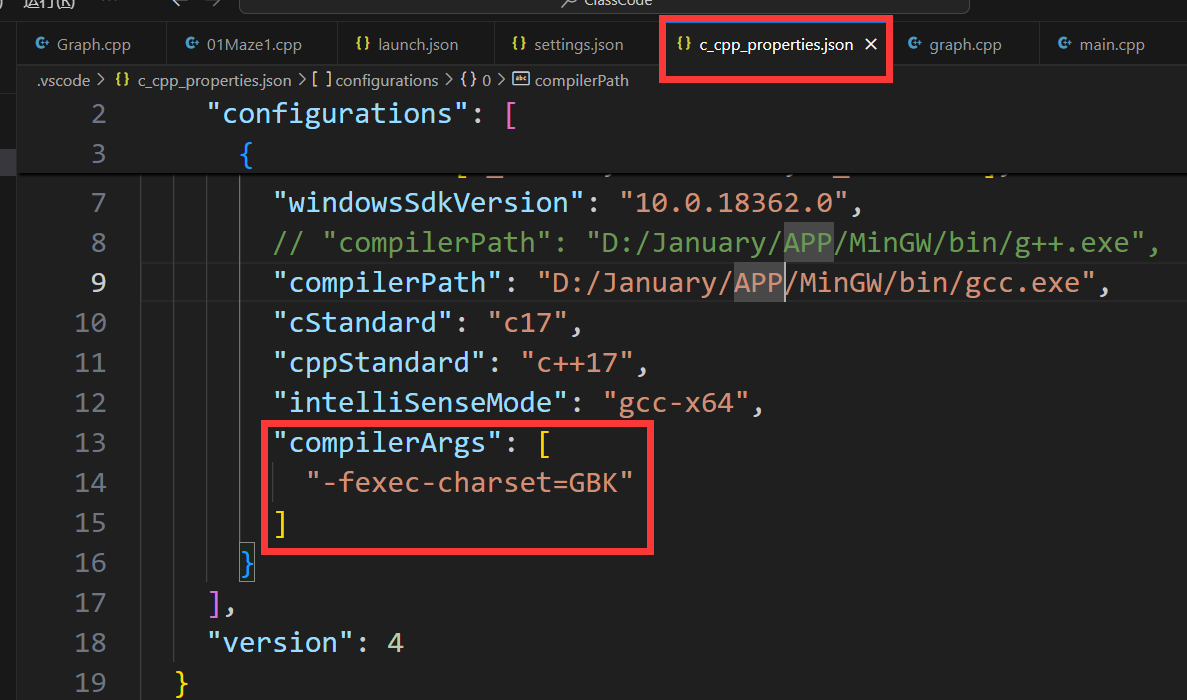


如果调试过程中在cmd输出的是乱码：按ctrl+shift+p打开控制面板，搜索C/C++:编辑配置(JSON)，按enter进入，会打开一个叫c\_cpp\_properties的json配置文件，加上以下配置即可

"compilerArgs": [

"-fexec-charset=GBK"

]



对于在输出端显示的中文乱码问题，至今都没有解决，不过可以通过在return处打断点的方式，使其在终端输出