项目3说明文档

数据结构课程设计

——勇闯迷宫游戏

作 者 姓 名： 谭欢秘

学 号： 1954185

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

目 录

[1 分析 1](#_Toc60165611)

[1.1 背景分析 1](#_Toc60165612)

[1.2 功能分析 1](#_Toc60165613)

[2 设计 2](#_Toc60165614)

[2.1 数据结构设计 2](#_Toc60165615)

[2.2 类结构设计 2](#_Toc60165616)

[2.3 成员与操作设计 3](#_Toc60165617)

[2.3.1 Stack类成员： 3](#_Toc60165618)

[2.3.2 Coordinate类成员： 4](#_Toc60165619)

[2.3.3 Maze类成员： 5](#_Toc60165620)

[2.4 主函数设计 5](#_Toc60165621)

[3 实现 6](#_Toc60165622)

[3.1 Stack类一些非核心功能的实现 6](#_Toc60165623)

[3.1.1 getSize()函数 6](#_Toc60165624)

[3.1.2 isEmpty()函数 6](#_Toc60165625)

[3.1.3 makeEmpty() 7](#_Toc60165626)

[3.1.4 getTop() 7](#_Toc60165627)

[3.2 Stack类元素进栈功能的实现 8](#_Toc60165628)

[3.2.1进栈功能流程图 8](#_Toc60165629)

[3.2.2进栈功能核心代码 8](#_Toc60165630)

[3.3 Stack类元素出栈功能的实现 9](#_Toc60165631)

[3.3.1元素出栈功能流程图 9](#_Toc60165632)

[3.3.2 元素出栈功能核心代码 9](#_Toc60165633)

[3.4 Stack类中查找元素是否在栈中功能的实现 10](#_Toc60165634)

[3.4.1 findEle()功能流程图 10](#_Toc60165635)

[3.4.2 findEle()功能核心代码 10](#_Toc60165636)

[3.5 Maze类中goUp()功能的实现 11](#_Toc60165637)

[3.5.1 goUp()功能流程图 11](#_Toc60165638)

[3.5.2 goUp()功能核心代码 12](#_Toc60165639)

[3.6 Maze类中intialize()功能的实现 13](#_Toc60165640)

[3.6.1 intialize()功能流程图 13](#_Toc60165641)

[3.6.2 intialize()功能核心代码 13](#_Toc60165642)

[3.7 Maze类中findPath()功能的实现 14](#_Toc60165643)

[3.7.1 findPath ()功能流程图 14](#_Toc60165644)

[3.7.2 findPath ()功能核心代码 14](#_Toc60165645)

[3.8 Maze类中showPath()功能的实现 16](#_Toc60165646)

[3.8.1 showPath ()功能流程图 16](#_Toc60165647)

[3.8.2 showPath ()功能核心代码 16](#_Toc60165648)

[3.9 主函数的实现 17](#_Toc60165649)

[3.9.1 主函数流程图 17](#_Toc60165650)

[3.9.2 总体系统核心代码 17](#_Toc60165651)

[3.9.3 总体系统截屏示例 18](#_Toc60165652)

[4 测试 18](#_Toc60165653)

[4.1 功能测试 18](#_Toc60165654)

[4.1.1 初始化功能测试 18](#_Toc60165655)

[4.1.2 寻找路径、打印路径功能测试 21](#_Toc60165656)

[4.2 边界测试 23](#_Toc60165657)

[4.2.1 迷宫大小数据非法 23](#_Toc60165658)

[4.2.2 迷宫起点终点数据非法 23](#_Toc60165659)

[5 总结 24](#_Toc60165660)

[5.1 遇到的错误 24](#_Toc60165661)

[5.1.1 xx函数中 24](#_Toc60165662)

[5.2 改进与优化 24](#_Toc60165663)

[5.3 项目心得 24](#_Toc60165664)

# 1 分析

## 1.1 背景分析

迷宫只有两个门，一个门叫入口，另一个门叫出口。一个骑士骑马从入口进入迷宫，迷宫设置很多障碍，骑士需要在迷宫中寻找通路以到达出口。在古希腊神话中，半人半牛的怪物弥诺陶洛斯（Minotaur）藏身于一个精心设计、结构极其复杂的迷宫之中。因此，找到并消灭它绝非易事，而此后如何顺利返回而不致困死更是一个难题。 不过，在公主阿里阿德涅(Ariadne)的帮助下，英雄忒修斯（Theseus）还是想出了个好办法， 他最终消灭了怪物，并带着公主轻松地走出迷宫。实际上，忒修斯所使用的法宝，只不过是一团普通的线绳。他将线绳的一端系在迷宫的入口处，而在此后不断检查各个角落的过程中，线团始终握在他的手中。线团或收或放，跟随着忒修斯穿梭于蜿蜒曲折的迷宫之中，确保他不致迷路。

忒修斯的高招，与现代计算机中求解很多问题的算法异曲同工，这正是试探回溯法。忒修斯真正实现这一方法的物质基础是他的线绳，而计算机实现试探回溯所依赖于的数据结构则是“栈”。

## 1.2 功能分析

路径规划是人工智能的基本问题之一，要求依照约定的行进规则，在具有特定几何结构的空间区域内，找到从起点到终点的一条通路。本项目应实现的功能有：用户指定一个空间区域限定为由 n \* n个方格组成的迷宫。迷宫中分布有若干障碍物，0表示可行、1表示为障碍物，迷宫障碍分布也由用户自行输入。最后计算机只通过水平或垂直移动，在用户任意指定的起点与终点之间找出一条通路。

由此可知，本项目应有初始化的功能——输入迷宫大小和障碍物分布、查询邻格是否为空的功能——若为空且不是来时的路径则转入、寻找路径的功能以及打印出路径的功能。

迷宫问题的求解过程可以采用回溯法即在一定的约束条件下试探地搜索前进，若前进中受阻，则及时回头纠正错误另择通路继续搜索的方法。从入口出发，按某一方向向前探索，若能走通，即某处可达，则到达新点，否则探索下一个方向；若所有的方向均没有通路，则沿原路返回前一点，换下一个方向再继续试探，直到所有可能的道路都探索到，或找到一条通路，或无路可走又返回入口点。在求解过程中，为了保证在达到某一个点后不能向前继续行走时，能正确返回前一个以便从下一个方向向前试探，则需要在试探过程中保存所能够达到的每个点的下标以及该点前进的方向，当找到出口时试探过程就结束了。

# 2 设计

## 2.1 数据结构设计

如上功能分析所述，迷宫寻径问题主要利用了回溯试探的方法：若前进中受阻，则及时回头纠正错误另择通路继续搜索的方法。从入口出发，按某一方向向前探索，若能走通，即某处可达，则到达新点，否则探索下一个方向；若所有的方向均没有通路，则沿原路返回前一点，换下一个方向再继续试探，直到所有可能的道路都探索到，或找到一条通路，或无路可走又返回入口点。在求解过程中，为了保证在达到某一个点后不能向前继续行走时，能正确返回前一个以便从下一个方向向前试探，则需要在试探过程中标记每一个经过的点。根据这一需求，我们应选择采用栈的数据结构。

## 2.2 类结构设计

因需要用栈，故使用一个模版类Stack<typename Type>完成。此外，完成迷宫主要需要一个迷宫类Maze，它有棋盘格用于存储每一个的元素，有完成初始化、寻找路径的函数；此外，还需要一个Coordinate类存储起点、终点以及当前查询的点，保存元素的横纵坐标与来时的方向。故Maze类应被设计为Coordinate类的友元。

## 2.3 成员与操作设计

### 2.3.1 Stack类成员：

### 2.3.2 Coordinate类成员：

因为Stack类中有些函数会涉及Type类的运算符，故需对Coordinate类的几个运算符进行重载。

### 2.3.3 Maze类成员：

其中Stack<Coordinate>类数据成员就是用来标记走过的路线的，因为是对Coordinate类做操作，所以Stack类模版在此的实现应为Coordinate类。需要注意的是，操作当前试探到的点移动的函数goUp()、goDown()、goLeft()、goRight()均使用引用作为参数，所以能直接改变current点，使其发生移动。

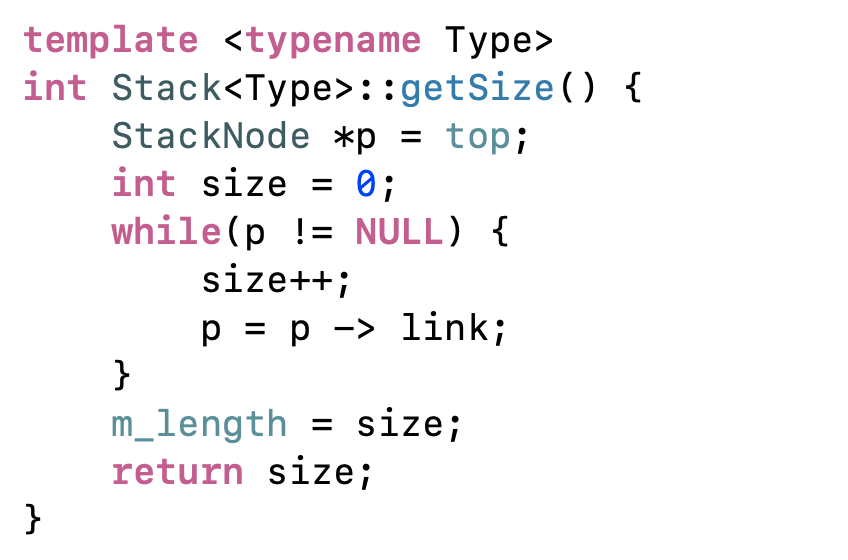
## 2.4 主函数设计

声明一个Maze类的对象maze。首先调用maze.intialize()函数实现迷宫的初始化，完成对迷宫的创建和输入障碍分布以及输入迷宫起点、终点；然后在intialize()中调用了showBoard()函数打印出迷宫盘；打印后调用maze.findPath()寻找迷宫路径，最后调用maze.showPath()打印出寻找到的路径。

# 3 实现

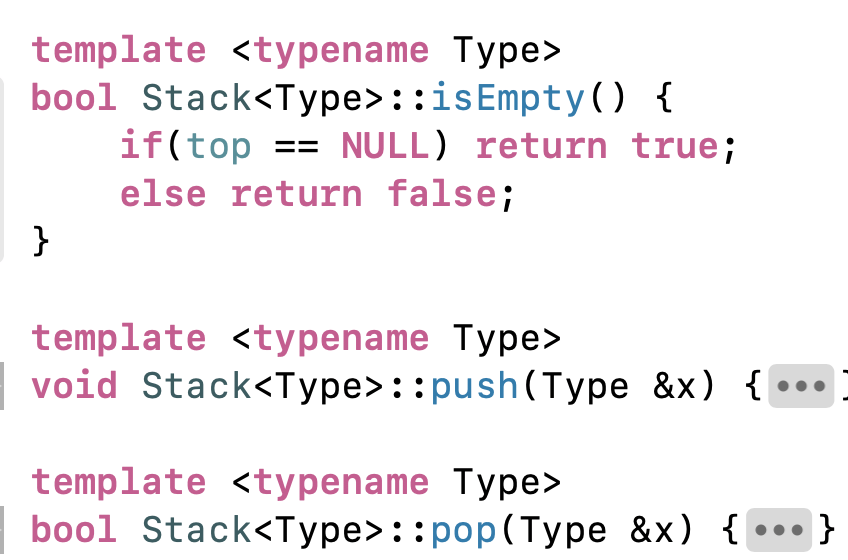
## 3.1 Stack类一些非核心功能的实现

### 3.1.1 getSize()函数



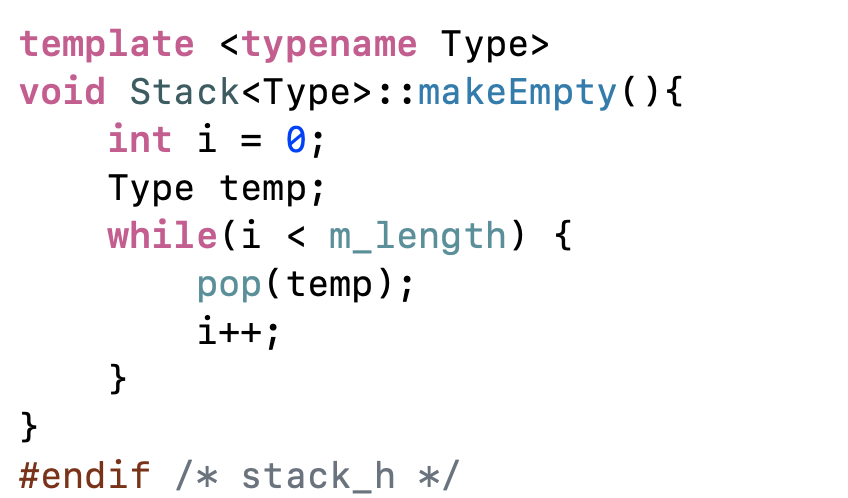
从栈顶元素开始，让指针p不断后移，每后移一次size++。直到p指向NULL时，循环结束，将size的值赋给Stack类的私有成员m\_length。每次执行了出栈、进栈操作后，可以利用该函数进行重新统计栈中元素个数；同时将size值设为函数return的值，在保证了Stack类封装性的前提下也让其他类可以读取栈中元素的总个数。

### 3.1.2 isEmpty()函数



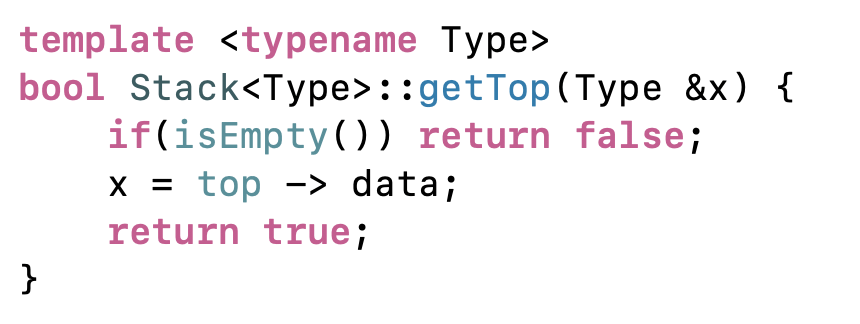
若栈顶元素指针为空，则表示栈空、返回true。

### 3.1.3 makeEmpty()



逐一将所有元素pop出栈即可。

### 3.1.4 getTop()

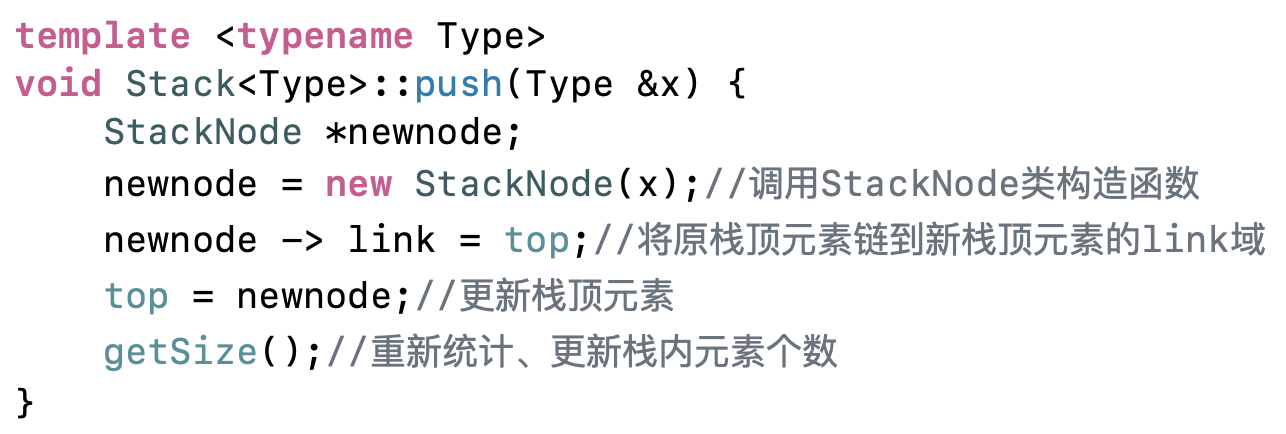


首先判断是否为空。若不为空，则取top的值赋给x，通过引用传递给参数。

## 3.2 Stack类元素进栈功能的实现

### 3.2.1进栈功能流程图

### 3.2.2进栈功能核心代码



## 3.3 Stack类元素出栈功能的实现

### 3.3.1元素出栈功能流程图

是

### 3.3.2 元素出栈功能核心代码

## 3.4 Stack类中查找元素是否在栈中功能的实现

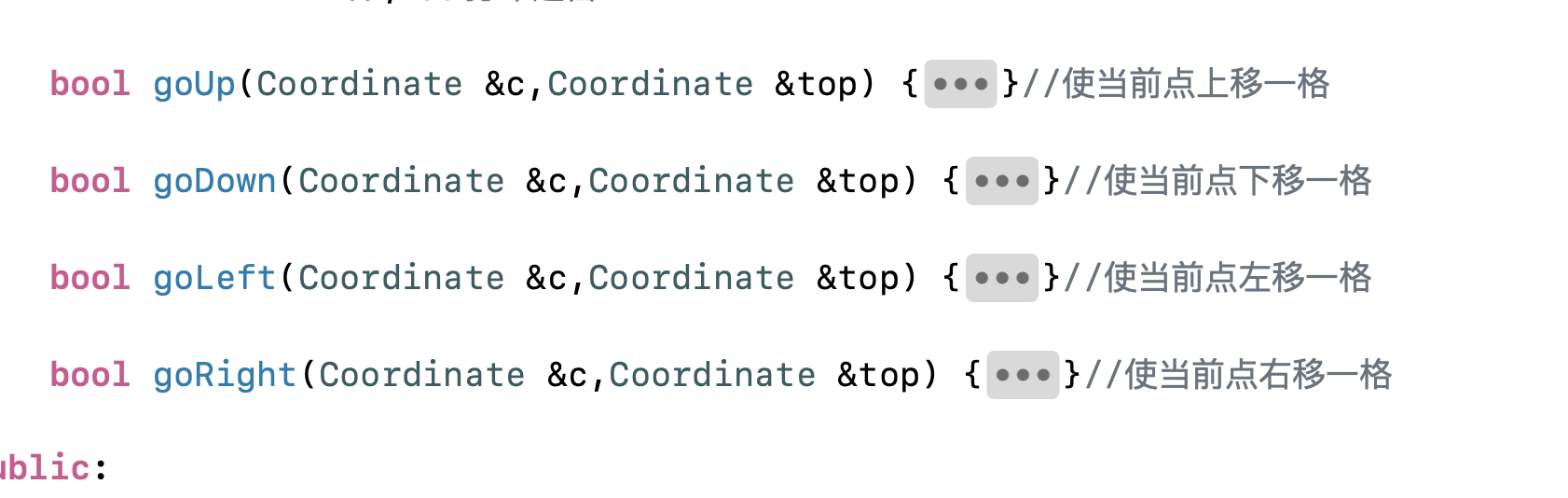
### 3.4.1 findEle()功能流程图

### 3.4.2 findEle()功能核心代码

## 3.5 Maze类中goUp()功能的实现

### 3.5.1 goUp()功能流程图

### 3.5.2 goUp()功能核心代码

同理可写类似的三个函数goDown()、goLeft()、goRight()，在此不再赘述了。

## 3.6 Maze类中intialize()功能的实现

### 3.6.1 intialize()功能流程图

### 3.6.2 intialize()功能核心代码

## 3.7 Maze类中findPath()功能的实现

### 3.7.1 findPath ()功能流程图

### 3.7.2 findPath ()功能核心代码

需要注意的是goUp() goDown()这些函数的参数是引用，所以current的值会被修改为移动之后的，所以pre要在新的current入栈前取栈顶元素为它赋值，然后再让修改后的current入栈。

## 3.8 Maze类中showPath()功能的实现

### 3.8.1 showPath ()功能流程图

### 3.8.2 showPath ()功能核心代码

需要注意的是，若直接用path栈输出，因为栈FILO（先进后出）的特性，打印出的路径将是从终点到起点的逆序路径，所以再利用一个辅助栈temp将path出栈的元素先存起来，以实现打印从起点到终点的正序路径。

## 3.9 主函数的实现

### 3.9.1 主函数流程图

### 3.9.2 总体系统核心代码

### 3.9.3 总体系统截屏示例

# 4 测试

## 4.1 功能测试

### 4.1.1 初始化功能测试

**测试用例1**：12 //迷宫大小

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

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

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

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

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

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

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

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

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

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

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

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

0 1 11 11 //迷宫起点、终点

**预期结果1**：

生成一个12\*12的迷宫，迷宫中障碍分布情况同输入。

**实验结果1：**

**测试用例2**：

20 //迷宫大小

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

0 1 19 18 //迷宫起点、终点

**预期结果2**：

生成一个20\*20的迷宫，迷宫中障碍分布情况同输入。

**实验结果2：**

### 4.1.2 寻找路径、打印路径功能测试

**测试用例1：**12 //迷宫大小

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

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

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

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

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

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

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

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

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

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

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

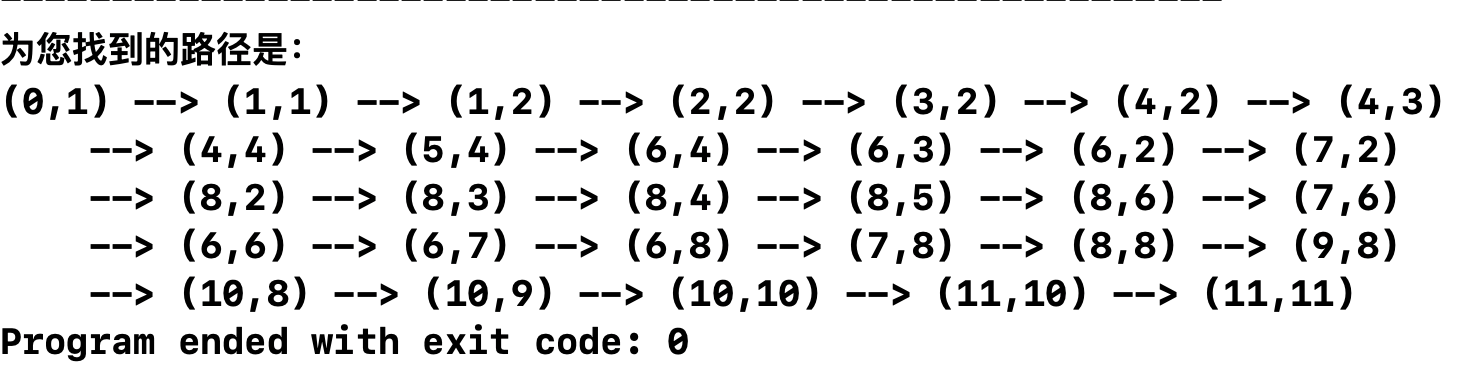
1 1 1 0 1 1 1 1 1 1 0 0

0 1 11 11 //迷宫起点、终点

**预期结果1：**

(0,1) --> (1,1) --> (1,2) --> (2,2) --> (3,2) --> (4,2) --> (4,3) --> (4,4) --> (5,4) --> (6,4) --> (6,3) --> (6,2) --> (7,2) --> (8,2) --> (8,3) --> (8,4) --> (8,5) --> (8,6) --> (7,6) --> (6,6) --> (6,7) --> (6,8) --> (7,8) --> (8,8) --> (9,8) --> (10,8) --> (10,9) --> (10,10) --> (11,10) --> (11,11)

**实验结果1：**

****

**测试用例2：**

20 //迷宫大小

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

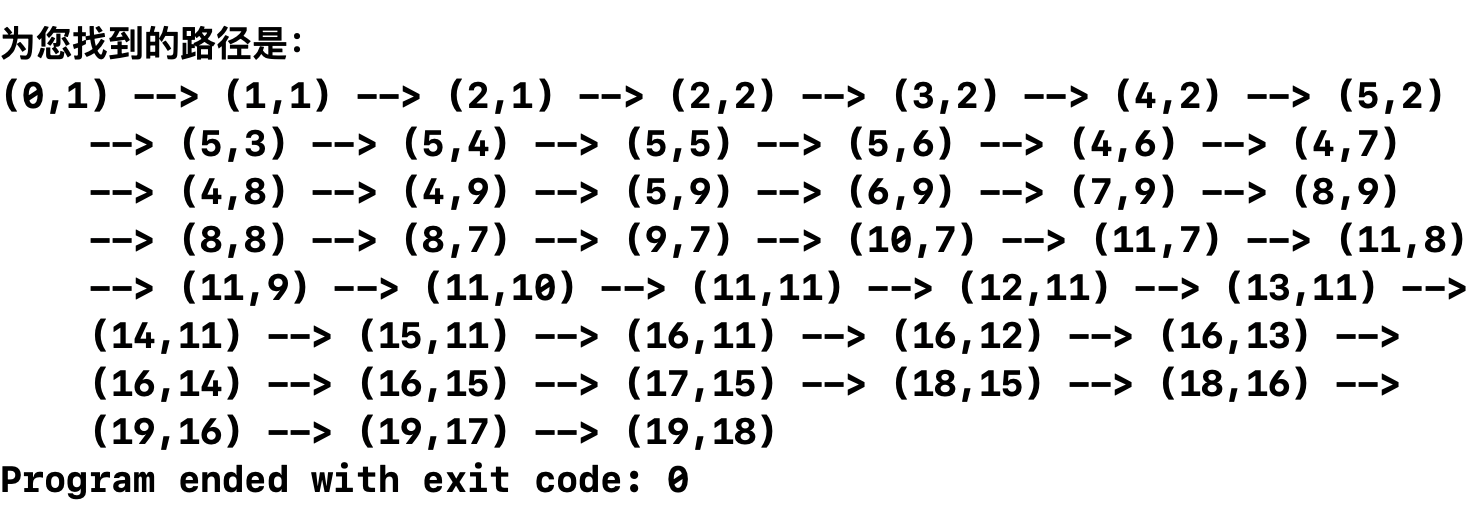
1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 1 1 1

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

0 1 19 18 //迷宫起点、终点

**预期结果2：**

(0,1) --> (1,1) --> (2,1) --> (2,2) --> (3,2) --> (4,2) --> (5,2) --> (5,3) --> (5,4) --> (5,5) --> (5,6) --> (4,6) --> (4,7) --> (4,8) --> (4,9) --> (5,9) --> (6,9) --> (7,9) --> (8,9) --> (8,8) --> (8,7) --> (9,7) --> (10,7) --> (11,7) --> (11,8) --> (11,9) --> (11,10) --> (11,11) --> (12,11) --> (13,11) --> (14,11) --> (15,11) --> (16,11) --> (16,12) --> (16,13) --> (16,14) --> (16,15) --> (17,15) --> (18,15) --> (18,16) --> (19,16) --> (19,17) --> (19,18)

**实验结果2：**

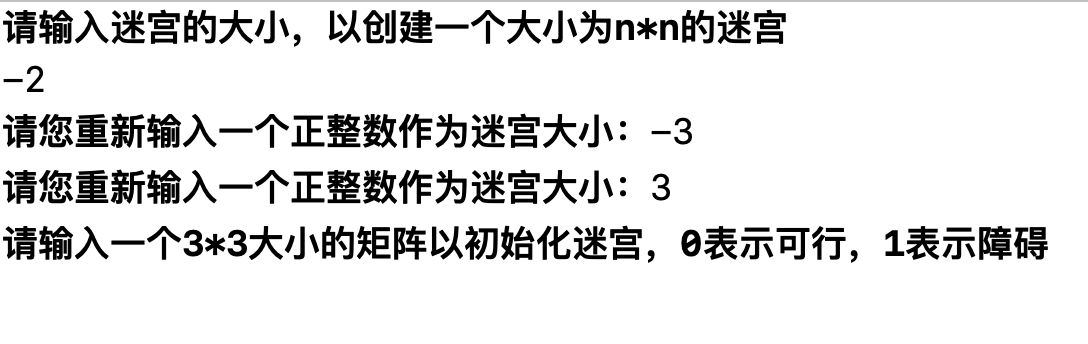
## 4.2 边界测试

### 4.2.1 迷宫大小数据非法

**测试用例：**初始输入迷宫大小数据非法，连续输入多个非正整数

**预期结果：**若输入非正整数，立即给出错误提示并要求重新进行输入，程序运行正

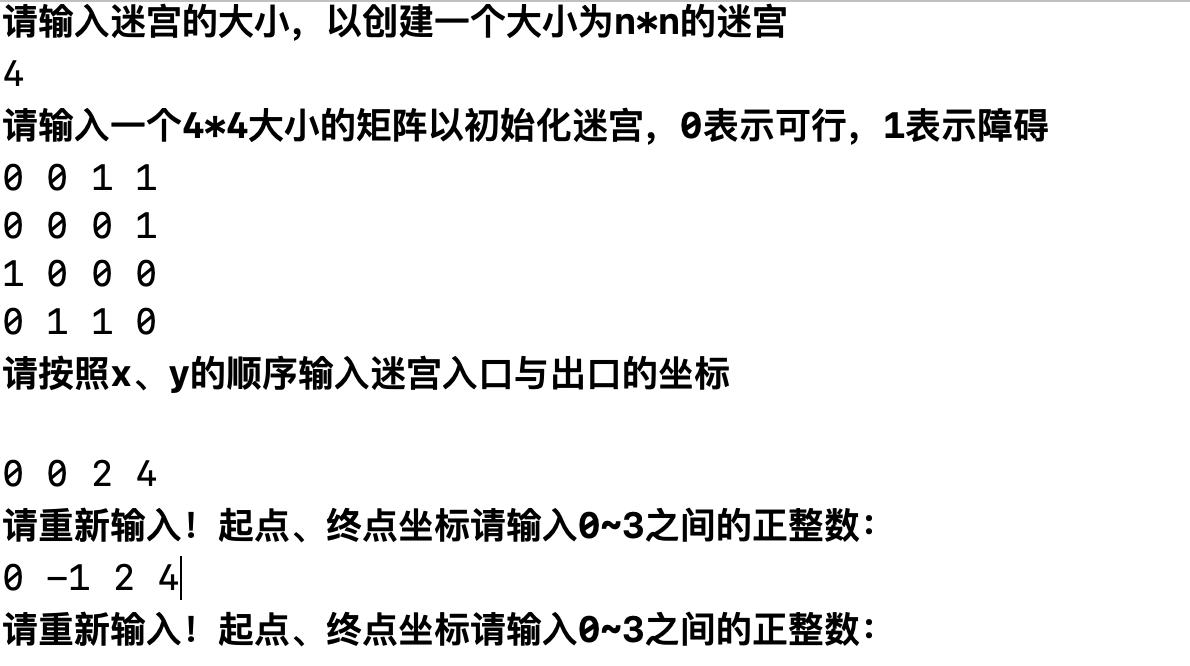
常不崩溃。

**实验结果：**

### 4.2.2 迷宫起点终点数据非法

**测试用例：**起点、终点坐标输入负数或越界

**预期结果：**程序提示输入不合法，要求重新输入直至合法，期间正常运行，不崩溃。

**实验结果：**

# 5 总结

## 5.1 遇到的错误

### 5.1.1 xx函数中

## 5.2 改进与优化

## 5.3 项目心得