项目说明文档

数据结构课程设计

——勇闯迷宫游戏

作 者 姓 名： 曹晓慈

学 号： 2052844

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

## 分析

### 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。

### 2.功能分析

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。

## 设计

### 数据结构设计

根据上面的功能分析，题目对输出的要求是从头输出，这里需要做到先进后出，故可以采用栈对路线进行输出。本题目选择使用链式栈对路线进行输出。而对于迷宫的储存则可以用数组进行，值为1时路为通路，否则则不通。

### 类结构设计

首先需要考虑到当一个结点到一个口的时候，需要对东西南北的四个方向进行判断，这里可以用struct Offsets来储存结点在水平方向和竖直方向的偏移量，用结构体Offsets的数组来储存四个方向的偏移。对于地图的储存可以用简单的二维数组进行储存。而对于链式栈，可以用StackNode和Stack 来表示。同时对于点的坐标还可以用Point的结构体来表示，还可以在这里对坐标点

### 成员与操作设计

struct Point

{

int x, y;

friend ostream& operator<<(ostream&, Point& Point) {//overload the operator <<

cout << " - - >(" << Point.x << "," << Point.y << ")";

return cout;

}

};

template <class Type> class Stack;

template <class Type> class StackNode

{

friend class Stack<Type>;

private:

Type data; //结点数据

StackNode<Type>\* link; //结点链指针

StackNode

(Type d = 0, StackNode<Type>\* l = NULL) : data(d), link(l) { }

};

template <class Type> class Stack

{

public:

Stack() : top(NULL) { }

Stack();

void Push

(const Type& item);

Type Pop();

Type GetTop();

void MakeEmpty(); //实现与

void OutStack();

int IsEmpty() const

{

return top == NULL;

}

private:

StackNode<Type>\* top; //栈顶指针

};

struct Offsets {

int a, b; // a和b是x,y方向的偏移

};

Offsets direction[8] = {

{-1, 0}, // 北：{-1, 0, "N"}

{0, 1}, // 东：{0, 1, "E"}

{1, 0}, // 南：{1, 0, "S"}

{0, -1}, // 西：{0, -1, "W"}

};

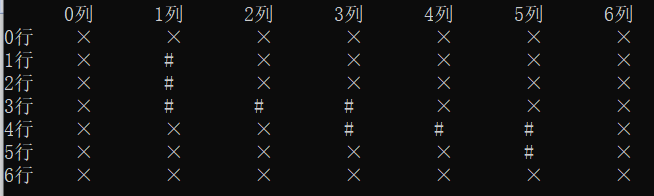
## 实现

### 地图实现

该游戏因要求将地图写死，故可将存储地图的二维数组作为全局变量，在建立表示地图的二维数组时即可将出路和死路写好，并根据数组进行地图的打印。

maze[1][1] = maze[2][1] = maze[3][1] = maze[3][2] = maze[3][3] = maze[4][3] = maze[4][4] = maze[4][5] = maze[5][5] = 1; //出路

in\_x=1,in\_y=1,out\_x = 5, out\_y = 5; //入口和出口



### 回溯算法实现

探索与回溯法）是一种选优搜索法，又称为试探法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为“回溯点”。

这里回溯算法可以用递归实现，对当前结点的四个方向进行判断，若可以走则继续进行递归，利用这种方式直到找到出路递归结束。这里还加入一个二维mark数组以对已经走过的路进行标记，若走过则标记为1避免重复而陷入的死循环。

bool seekpath(int x,int y,Stack<Point>& OUT)

{

if (x == out\_x && y == out\_y)

{

return true;

}

else

{

for (int i = 0; i < 8; i++)

{

x = x + direction[i].a;

y = y + direction[i].b;

if (maze[x][y] == 1 && mark[x][y] == 0)

{

mark[x][y] = 1;

if (seekpath(x, y, OUT))

{

Point tmp = { x,y };

OUT.Push(tmp);

return true;

}

}

else

{

x = x - direction[i].a;;

y = y - direction[i].b;

}

}

}

return false;

}

### 利用栈对输出的实现

由于在算法寻找迷宫出口的过程中，只有到达迷宫出口时才会返回，因此如果在递归过程中进行输出，得到的迷宫路径是逆向的，因此这里采用栈让先得到的坐标到栈底，以达到先入后出的效果。

template <class Type> void Stack<Type>::Push(const Type& item)

{

top = new StackNode<Type>(item, top);

//新结点链入top之前, 并成为新栈顶

}

template <class Type> Type Stack<Type>::Pop()

{

assert(!IsEmpty());

StackNode<Type>\* p = top;

Type retvalue = p->data; //暂存栈顶数据

top = top->link; //修改栈顶指针

delete p;

return retvalue; //释放, 返回数据

}

void Stack<Type>::OutStack()

{

while (top != NULL)

{

Type tmp = Pop();

cout << tmp;

}

}

### 主要系统实现

对于main函数，由于地图是写死的，所以在main函数中已经将迷宫的出路写在maze的二维数组中，并用for循环进行迷宫的格式化输出，后面再调用seekpath()来寻找迷宫的出路。

核心代码：

Stack<Point> Out;

cout << "迷宫地图" << endl;

maze[1][1] = maze[2][1] = maze[3][1] = maze[3][2] = maze[3][3] = maze[4][3] = maze[4][4] = maze[4][5] = maze[5][5] = 1;

in\_x=1,in\_y=1,out\_x = 5, out\_y = 5;

int x = in\_x, y = in\_y;

mark[x][y] = 1;

for (int i = 0; i <= 6; i++)

{

cout << " "<<i<<"列";

}

cout << endl;

for (int i = 0; i <= 6; i++)

{

cout << i << "行 ";

for (int j = 0; j <= 6; j++)

{

if (maze[i][j] == 1)

cout << "# ";

else

cout << "× ";

}

cout << endl;

}

cout << endl;

cout << "迷宫路径" << endl;

cout << endl;

seekpath(x, y,Out);

## 测试

因该题无输入和输出，所以直接放项目运行截图。

