자료구조 MAZE

20171666 고재원

우선 조건으로부터 우리가 생각해야 할 것들을 적어보기로 한다.

(a) mark[][] 배열을 사용하지 않는다.

mark 배열은 지나온 공간의 행, 열을 저장하는 용도이다. 하지만 이것을 쓰지 말아야 한다. 그렇다면 지나온 길을 표시할 때 쓸 수 있는 다른 저장 장소가 있을까? 라는 생각을 가지며 내가 가지고 있는 변수들을 살펴보았다. 그러던 도중, Maze 행렬이 눈에 들어왔다. 미로의 0 값이 우리가 지나가는 장소이다. 그러면 이 미로에다가 지나온 길을 표시한다면? 예를 들어 장소를 지나가면 0에서 1로 새로 저장하여 지나온 길을 체크하는 것 이다. 우선 디테일은 나중에 생각하고, mark배열 대신 maze를 쓰기로 하였다.

(b) maze[][] 배열의 상하좌우 가장자리 element들을 사용하지 않는다. 즉, n by m maze의 경우 교재의 프로그램에서는 maze[][] 배열을 maze[n+2][m+2] 로 선언하고 상하좌우 가장자리 element들의 값을 1로 설정하여 사용하는데 본 수정에서는 maze[][] 배열을 maze[n][m]로 선언하여 maze를 표현한다.

상하좌우에 1이 없다면, 만약 다음 길을 찾을 때 가능한 경우의 수 들이 범위를 넘어서는지 조건을 하나 더 추가해야 된다. 만약 Maze[0][8]이면, 바깥으로 나갈 수 있는 경우를 생각해서 이를 제외해야 한다. 이 범위를 체크하여 올바른 길 만 찾아주는 함수를 생각 해두자.

(c) stack에 저장 시 위치좌표만 저장하고 방향정보는 저장하지 않는다. 즉, 교재에서는 (row, col, dir)을 push하는데 (row, col) 만 push한다.

dir이 없으면 갈 방향을 알 수 없게 된다. 따라서 매우 중요한 정보이다. 하지만 stack에서 쓸 수 없다면 이동 가능한 길이 존재하는지, 그 좌표는 무엇인지 정도를 판단 가능한 함수를 새로 만들어서 처리를 해야 될 것 같다.

(d) 출구좌표에 도달했는지의 체크를 현재 위치좌표가 출구좌표와 동일한지를 확인하는 것으로 수행하도록 한다. 교재에서는 현재 위치좌표가 출구좌표의 바로 이웃 위치좌표인가를 확인하는 것으로 수행하고 있다.

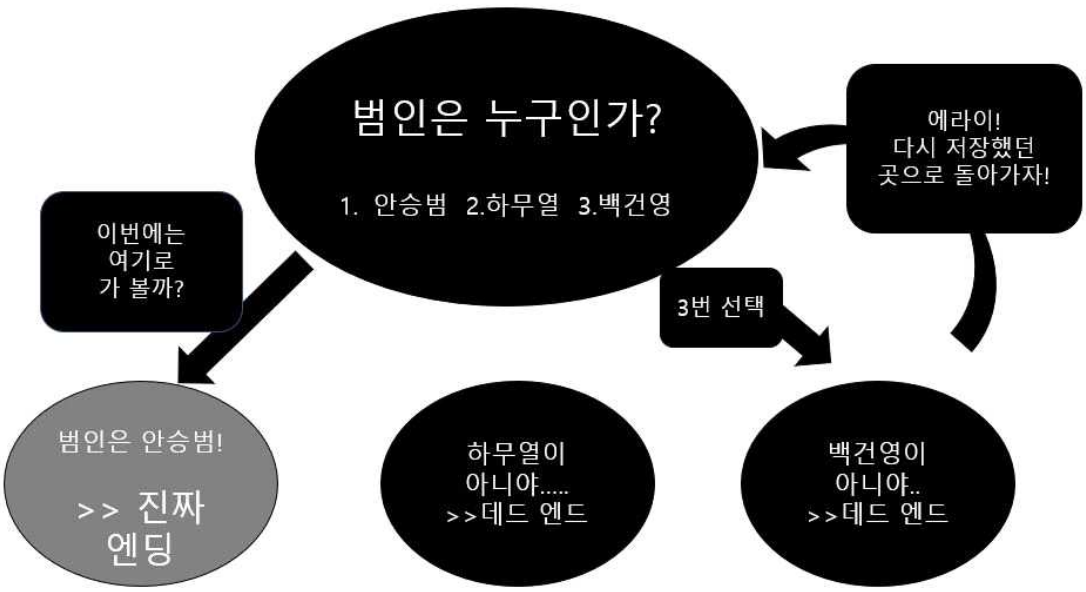
출구 좌표에 도착 할 때까지 프로그램이 실행되어야 한다는 것을 알려주는 것 같다. 크게 변한 것은 없는 것 같다.

하지만, 제일 중요한 것은 이것이다.
“그래서, 절차(북쪽 먼저, 시계방향으로)를 밟아서 길을 찾는데 막다른 길이면 어떻게 되돌아갈래? 지나온 길을 표현하는 저장장소 없이 말이야.”

물론 절반은 해결된 의문이다. Maze에 지나온 길을 표시하기로 하였으니 이제 어떻게? 하는 것만 남았다. 이때, 과거의 기억이 뇌를 스쳐갔다.

어렸을 때 유행했던 ‘검은방’이란 게임을 아는가? 게임에서 선택지가 주어지는데, 이때 결정한 선택지에 따라 각기 다른 엔딩을 맞이한다. 각 장면에서 저장을 할 수 있으며, 나중에 불러오기도 할 수 있었다. 그런데 왜 갑자기 중간고사 과제에 게임 얘기를 하냐고 의문이 들 수 있다. 바로 이 게임의 구조에서 힌트를 찾았기 때문이다.

게임에서 우리가 원하는 것은 진짜 엔딩이다. 수많은 가짜 엔딩(ex 주인공이 죽는다.) 사이에서 우리는 진짜 엔딩에 도달하기 위해 선택을 잘 해야 한다. 만약 선택지가 4개라면, 그 중 한 개만 진짜 엔딩으로 갈 수 있는 선택지이고 나머지는 가짜, 즉 막다른 길 일 것 이다. 물론 선택을 잘 해서 진짜 엔딩에 도달할 수 있지만, 대개는 잘못된 엔딩을 맞이하게 된다. 이때 필요한 것이 무엇인가? 바로 저장/불러오기 기능이다. 선택을 해야 될 때, 그 상황을 저장한다. 그리고 선택을 잘못하여 데드 엔드가 되었을 경우, 우리는 그저 불러오기를 하여 처음 선택했던 그 장면으로 되돌아가면 된다.



만약 선택지가 없는 장면이라면 그냥 주어진 길을 따라 지나가면 된다. 우리는 그저 선택지가 있는 장면에서 저장, 불러오기를 하면 된다.

이제 게임의 장면을 미로의 한 칸, 각 선택지가 있는 장면은 여러 개의 다음 길이 있는 칸, 가짜 엔딩은 데드 엔드, 진짜 엔딩이 출구라고 생각하자. 그렇다면 이렇게 미로 찾기를 할 수 있다.

'우선 쭉 외길을 따라가다가 갈림길이 나오면 이 칸을 저장한다. 그 뒤 절차적으로 택한 길을 찾아 가다가 데드 엔드(주변에 길이 지나온 길 밖에 없다)면 최근의 갈림길로 다시 되돌아간다. 만약 모든 갈림길을 갔는데 데드 엔드면 그 이전의 갈림길로 되돌아간다.'

이러다 보면 우리는 진짜 엔딩, 즉 출구에 도달 할 수 있다. 이때 생각 할 것이 있다. 지나온 길과 갈림길을 같은 녀석으로 취급하되, 갈림길은 따로 특별함을 주어 나중에 갈림길로 되돌아갈 때 유용하게 쓰일 수 있어야 한다. 만약 지나온 길은 1로 저장하고, 갈림길은 2로 저장하면 어떨까?

Ex) 주어진 미로에서 (5,1)은 갈림길 없이 외길을 따라 가면 되므로 maze[5][1]에는 1을 저장한다. 하지만 (6,2)에서는 갈림길 (7,3),(7,2),(7,1)이 존재하므로 maze[6][2]에는 2를 저장한다. 그래서 나중에 길을 잃더라도 2가 저장된 최근 위치로 되돌아 올 수 있다. (즉 둘 다 0은 아니므로 현재 위치에서 다음 길 체크 시 제외된다.)

또한 이 길 찾기의 핵심 방법은 바로 스택이다. 스택에 길을 하나하나 쌓아가는 것이다. 여기서 또 기가 막힌 아이디어를 생각하였다.
내가 가는 길이 올바른 길이라고 가정 하고 스택에 막 절차적으로 가는 길을 쌓는다. 물론 착실히 maze[][]의 값을 1,2로 갱신하면서. 그러다가 데드 엔드를 맞이하게 된다. 이때, 진정하고 stack의 top에서부터 조사를 시작한다. 최근의 갈림길을 가지고 있던 그 위치 좌표를 찾는 것 이다.

확실한 것은 내가 데드 앤드에 도달했을 시 데드 엔드의 좌표와 최근 갈림길 좌표까지의 path(이 path에서 갈림길 좌표는 제외)는 모두 출구와 연관이 없는, 즉 스택에서 제외되어야 될 좌표들이라는 것이다.

따라서 만약 데드 엔드일시, 스택에서 최근의 갈림길 좌표까지를 제거(pop)해가면 된다. 다시 갈림길로 돌아왔을 시, 저번에 갔던 길은 이미 1처리가 되어 있어 다시 갈림길에서 길 체크를 해도 제외 될 것이다.

ex) 주어진 미로의 (2,2)에서, 절차 순으로 (3,3)으로 갔지만 (4,3)에서 데드 엔드를 맞이하게 되었다. 따라서 (3,3),,,,(4,3)은 잘못 된 길이므로 스택에서 사라져야 된다. 그렇다면 스택에 저장되어있던 (0,0),,,,,,(2,2),(3,3),(4,3)에서 최근 갈림길인 (2,2)를 찾기 전까지 저장된 값 (3,3),(4,3)을 Pop 하면 된다. 그 뒤 (2,2)에서 (3,3),(3,1) 두 개의 길을 선택 할 수 있는데 (3,3)은 이미 갔다 온 길이라 제외된다. 따라서 절차를 따라 자동적으로 (3,1)을 선택하여 길을 간다.

즉 이를 수도 코드로 만들면 다음과 같다. 기타 디테일은 다음에 다루도록 한다.

x= 현재 위치의 x좌표 y = 현재 위치의 y좌표

```
==========================================================================
while(1):
    if (  현재좌표 == 출구 좌표):
        종료;
    else:
        길 찾기(x,y)
        새로운 x,y좌표 = 스택의 맨 위의 값
==========================================================================
```

이제 길을 찾는 디테일을 알아보자.

우선 (x,y)좌표를 함수에서 받았을 때, 갈 수 있는 길을 검사해야 된다. 이때 검사하는 칸은 모두 8칸, 북쪽을 시작으로 시계 방향 순으로 조사한다. 이때, 배열의 범위를 넘어가거나 0보다 큰 값을 가진 칸은 갈 수 없으므로 제외해야 된다.

	y-1	y	y+1
x-1	7	0	1
x	6	현재좌표	2
x+1	5	4	3

8칸짜리 배열을 만들어 칸에 적힌 숫자의 순서대로 길이 있는지 없는지 저장하도록 하자. 만약 길이 있으면 1, 없으면(범위 넘어가는 것도 여기에 포함) 0이라고 저장하자.
주어진 미로의 (2,2)를 예시로 들면

(2,2)	0	1	2	3	4	5	6	7
좌표	1,2	1,3	2,3	3,3	3,2	3,1	2,1	1,1
값	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**

이 배열로부터 우리는

1. 현재 좌표에서의 다음으로 갈 수 있는 길의 개수
2. 길이 있으면 다음 길은 어디인가

를 알 수 있다. 여기로부터 길 찾기의 방법을 이어 나갈 수 있다. 방금 주변을 검사하여 배열에 저장하는 방법을 **'모든 길 찾기'**라고 하자.

==========

길 찾기(x,y):

  모든 길 찾기 실행

  if (모든 길 찾기에서 길의 개수 == 1):

        maze[현재위치] = 1 // 외길이다. 저장을 꼭 하자.

        다음 좌표 = 그 길

  else if(모든 길 찾기에서 길의 개수 > 1):

        maze[현재위치] = 2 // 갈림길이다.

        다음 좌표 = **절차적으로 다음 길 찾기**

  else

        maze[현재위치] = 1 // 데드 엔드. 그래도 저장은 해 주자

                                // 데드 엔드이므로 이때 스택에서 하나씩 pop해보자.

        while(maze[스택의 top이 가리키는 좌표] == 2): // 최근 갈림길까지 pop

              pop(stack)

==========

< 절차적으로 다음 길 찾기 >

주어진 좌표에 길이 존재할 때다. 배열로 값을 받으면 계산이 편하다. 우선 배열에서 0~7까지 북, 북동, ,,순의 절차적으로 저장되어있기 때문에 index 0부터 검사를 하여 1이 나오는 첫 번째 항이 다음 좌표이다. 아까 예시로 든 (2,2)의 배열을 보면 처음 1이 나타나는 항은 [3] 즉 (3,3) 이며, 이 좌표가 다음 좌표가 됨을 알 수 있다.

즉 절차적으로 다음 길 찾기는 다음과 같다.

==========

절차적으로 다음 길 찾기 (배열, x, y){

        for(t =0 ; t < 8 : t++): // 다음 길을 절차적으로 찾는다.

              if(배열[t] == 1)

                    FirstRoad = t

                    break;

```
        switch(FirstRoad){ // 그 값에 맞는 좌표를 찾는다.
                case 0 : 다음 좌표 = 북 쪽
                case 1 : 다음 좌표 : 북동 쪽
                case 2 : 다음 좌표 : 동 쪽
                        .............
                case 7 : 다음 좌표 : 서북 쪽
        }
}
```

==========

이제 모든 디테일은 얼추 완성 되었다. 이것들을 중심으로 코딩을 해 보자. 추가로 이전에 배웠던 스택에 관한 함수 중 무엇을 쓸지 생각해보자.

기본적으로 pop, push 가 쓰인다. 더 생각해보니 이 둘로도 충분하다.
이를 이용하여 짠 코드는 다음과 같다.

```
#include <stdio.h>
#include <stdlib.h>

#define Stack_Size 100
#define numRow 10
#define numCol 10

typedef struct {
        short int x;
        short int y;
} element; // 스택에 dir의 정보를 넣지 않았음

typedef struct {
        element storage[Stack_Size];
        int top;
}Stack;// 스택에 dir의 정보를 넣지 않았음


int maze[numRow][numCol] = {
        { 0,0,1,0,1,1,1,0,1,0 },
        { 1,0,0,1,1,1,0,1,0,1 },
        { 1,1,0,1,1,0,1,0,1,1 },
        { 0,0,1,0,1,1,1,0,0,0 },
        { 0,1,1,0,1,0,1,0,1,0 },
        { 1,0,1,1,1,1,0,0,1,0 },
        { 1,1,0,1,0,1,0,0,1,0 },
        { 1,0,0,0,1,0,1,0,0,0 },
        { 0,1,0,1,1,1,0,1,1,0 },
        { 1,0,0,1,1,1,0,0,0,0 }
};// maze의 주변을 1로 둘러싸지 않았음


void push(Stack* e, element storage) {
        e->top++;
        e->storage[e->top].x = storage.x;
        e->storage[e->top].y = storage.y;
}
```

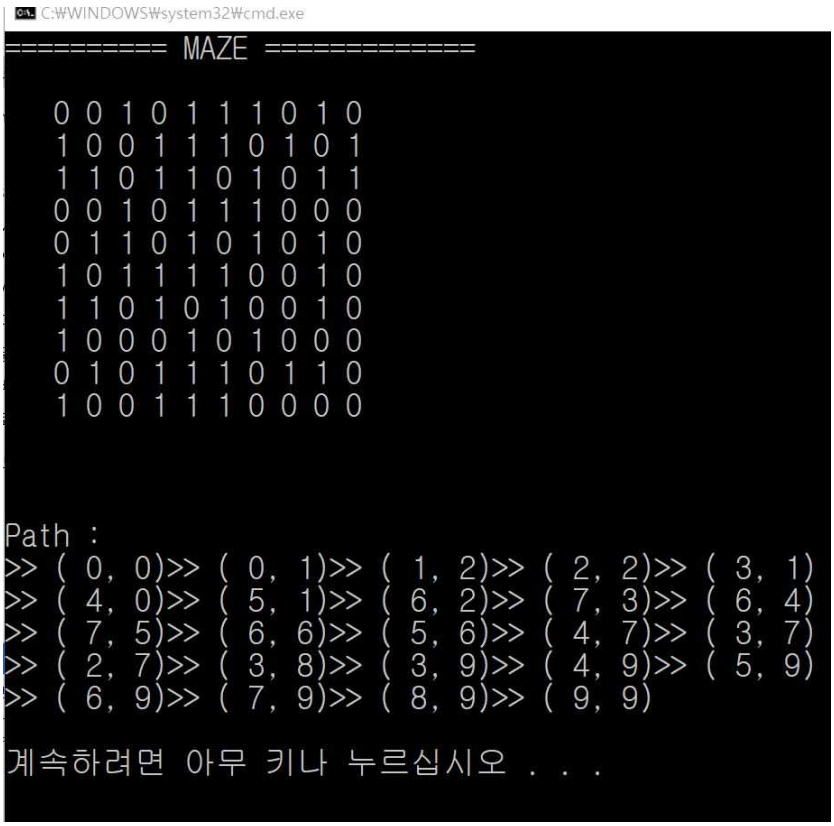
```
element pop(Stack* e) {
	if (e->top >= 0)
		return e->storage[(e->top)--];
	else {
		printf("stack is empty....₩n");
		exit(1);
	}
}

int Check_Range(int x, int y) {
	// 받은 점의 좌표가 행렬을 벗어나는지 확인
	if ((0 <= x) && (x < numRow) && (0 <= y) && (y < numCol))
		return 1;
	else return 0;
}


int check_roadNum(int* arr) {
	// 그 점에서의 다음에 갈 길의 개수 반환
	int HowManyRoads = 0;

	for (int i = 0; i < 8; i++)
		if (arr[i] == 1)
			HowManyRoads++;

	return HowManyRoads;
}

element Find_next_road(int* arr, int x, int y) {
	// 북쪽 먼저 시계방향 순으로 탐색 시 먼저 탐색하게 되는 다음 점 반환

	int FirstRoad = 0;
	element result_loc;

	for (int i = 0; i < 8; i++) {
		if (arr[i] == 1) {
			FirstRoad = i;
			break;// 절차적으로 첫 번째 길 찾기
		}
	}
	switch(FirstRoad){//그 첫 번째 길의 좌표 찾기
		case 0: result_loc.x = x - 1;  result_loc.y = y;      break; // 북
		case 1: result_loc.x = x - 1;  result_loc.y = y + 1;  break; // 북동
		case 2: result_loc.x = x;      result_loc.y = y + 1;  break; // 동
		case 3: result_loc.x = x + 1;  result_loc.y = y + 1;  break; // 동남
		case 4: result_loc.x = x + 1;  result_loc.y = y;      break; // 남
		case 5: result_loc.x = x + 1;  result_loc.y = y - 1;  break; // 남서
		case 6: result_loc.x = x;      result_loc.y = y - 1;  break; // 서
		case 7: result_loc.x = x - 1;  result_loc.y = y - 1;  break; // 서북
	}
	return result_loc;
}

void Find_every_road(int* arr, int x, int y ) {
	// 인접한 모든 방향에 길이 있는지만 탐색(범위 고려)
	// arr에는 모두 0이 저장되어 있음.
	if (Check_Range(x-1, y) == 1) {
		if (maze[x-1][y] == 0)
			arr[0] = 1;// 북 // 길이 있으면 해당 항에 1 저장.
		}
	if (Check_Range(x-1, y+1) == 1) {
		if (maze[x-1][y+1] == 0)
			arr[1] = 1;//북동
		}
	if (Check_Range(x, y+1) == 1) {
		if (maze[x][y+1] == 0)
			arr[2] = 1;//동
		}
```

```
        if (Check_Range(x+1, y+1) == 1) {
                if (maze[x+1][y+1] == 0)
                        arr[3] = 1;//동남
        }
        if (Check_Range(x+1, y) == 1) {
                if (maze[x+1][y] == 0)
                        arr[4] = 1;//남
        }
        if (Check_Range(x+1, y-1) == 1) {
                if (maze[x+1][y-1] == 0)
                        arr[5] = 1;//남서
        }
        if (Check_Range(x, y-1) == 1) {
                if (maze[x][y-1] == 0)
                        arr[6] = 1;//서
        }

        if (Check_Range(x-1, y-1) == 1) {
                if (maze[x-1][y-1] == 0)
                        arr[7] = 1;//서북
        }

}

void Find_Path(Stack* s, int x, int y) {
        // 가장 중요한 함수. 여기서 길을 찾는다.
        int WhereToGo[8] = { 0,0,0,0,0,0,0,0 };//여기에 길 북,북동,동 순으로저장
        element next_Location;// 여기에 다음 갈 길 저장
        element tmp;

        Find_every_road(WhereToGo,x,y); // 이 점 주변에서의 모든 길을 찾는다.


        if (check_roadNum(WhereToGo) == 1) {// 즉 길이 1개만 있을 때
                maze[x][y] = 1;//외길이다
                next_Location = Find_next_road(WhereToGo, x, y);
                push(s, next_Location);
        }
        else if (check_roadNum(WhereToGo) > 1) {// 길이 여러개 일때
                maze[x][y] = 2;  //  이 점에서 길이 여러개이므로 이 점이 체크포인트이다.
                next_Location = Find_next_road(WhereToGo, x, y);
                push(s, next_Location);
        }

        else { // 길이 없다 ㅜ
                maze[x][y] = 1; // 그래도 maze에 지나온 길 저장.
                tmp = pop(s);
                while (maze[tmp.x][tmp.y] == 2) // 최근 체크 포인트로 돌아간다.
                        tmp = pop(s);// 스택에 저장된 잘못 된 길을 지운다.
        }
}



int main() {
        element location;
        Stack s;

        s.top = -1; // 스택을 비운다.

        location.x = 0;
        location.y = 0; // 시작점의 좌표 대입

        push(&s, location); // 스택에 저장

        printf("========== MAZE ============₩n₩n");// Maze 출력

        for (int z = 0; z < numRow; z++) {
```

```
                printf("    ");
                for (int v = 0; v < numCol; v++) {
                        printf("%d ", maze[z][v]);
                }
                printf("₩n");
        }
        printf("₩n");// end 출력



        while (1) {
                if ((location.x == numRow - 1 && location.y == numCol - 1))
                        break;// 즉 maze[9][9]에 도착하면 종료. 조건에 만족

                else if (s.top < 0) {
                        printf("Can't find the way out..... ₩n");
                return 0; //출구가 없어 스택에 아무 값도 저장이 안 되어 있을 때
                }

                else {
                        Find_Path(&s, location.x, location.y);
                        location = s.storage[s.top]; // 맨 위의 스택의 값 복사
                }
        }

        printf("₩n₩n");
        printf("Path :₩n");// 경로 출력

        for (int n = 0; n <= s.top; n++) {
                printf(">> (%2d,%2d)", s.storage[n].x, s.storage[n].y);
                if ((n+1) % 5 == 0) //  5자리씩 끊어서 출력
                        printf("₩n");
        }
        printf("₩n₩n");
        return 0;
}
```

==========

또한 이 프로그램의 결과는 다음과 같다.

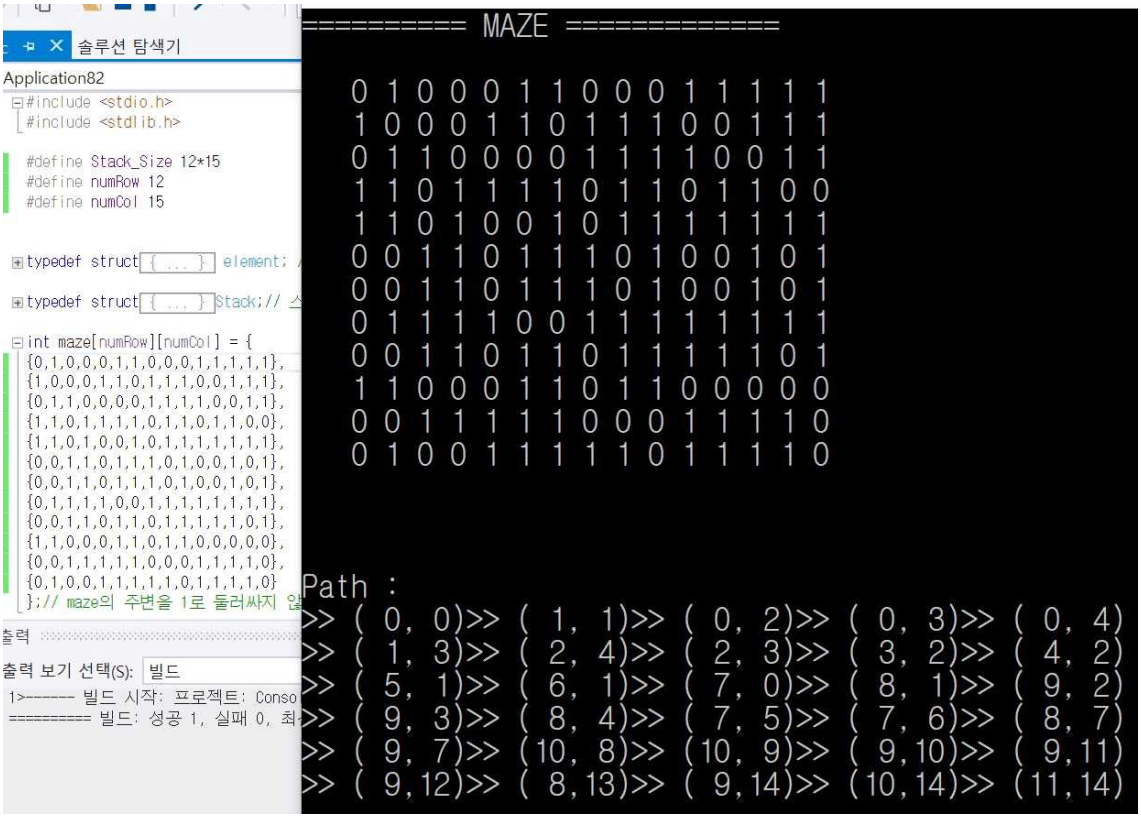


사진에서 Path도 주어진 절차대로 올바르게 길을 찾아 나서는 것을 알 수 있다. 또한 Mark[][]를 사용하지 않았으며, 미로 주변에 1을 둘러싸지 않았고, stack의 storage에는 row와 col에 대한 정보만 저장하였다. 마지막으로 현재 위치가 출구인지 검사하여 맞을 시 종료하게 만들었다.

하지만 여기서 끝내지 말고 테스트를 하나 더 해보자. 코드 검사 시 rowNum, colNum, Maze[][]를 바꿔서 테스트 한다고 하였다. 과연 이 코드가 변경 후에도 문제없이 돌아갈까?

그래서 번역본 기준 130p 3.8의 예제 미로로 실험을 하기 로 하였다.
즉 rowNum = 12, colNum = 15, maze[][] = 많아서 생략, 그림 참조.

위 사항을 변경 한 후 코드를 돌린 결과는 다음과 같다.



이전과 다른 문제없이 길을 찾아내는 모습을 볼 수 있다.

<프로그램을 만들면서...>

다행히도 아이디어가 번득 떠올라서 메인 아이디어를 코딩할 때는 큰 어려움은 없었다. 하지만, 현재 위치에서 주변이 범위를 넘어서는지, 길이 있는지, 그 중 어느 위치를 절차적으로 먼저 가게 되냐를 판단하는 함수가 생각보다 조건이 많고 까다로웠다. 주변이 범위를 넘어서는지 판단하는 것은 함수(Check_Range)로 만들어 필요할 때마다 호출하였다. 이 부분을 만들면서 조금 더 간단한 방법은 없을까 라는 아쉬운 마음이 들었다.