Computer Science 102
Lab 12

In this lab you will you will complete the maze.c program. The program uses C++ input and output so it must be compiled with g++, but it uses no C++ classes. You will have to complete three functions.

A maze problem

Consider the following two dimensional array

```
0 0 0 0 0 0
1 1 1 1 1 0
0 0 0 0 0 0
0 1 1 1 1 1
0 0 0 0 1 1
1 1 1 0 0 0
```

We can view this as a maze in which passage through locations have a value of 0 is permitted but passage through locations having a value of 1 is not. We add the additional constraint that movement is permitted between adjacent locations in the array in *only vertical and horizontal (not diagonal)* directions. As is conventional, the value *maze[0][4] represents row 0 column 4*.

Our mission will be given a start (row, col) and a target (row, col), print a path (if one exists) beween the start and the finish. For the maze shown here given a start (0, 0) and a target (5, 5) the path is:

```
0 0 0 0 0 0
1 1 1 1 1 0
0 0 0 0 0 0
0 1 1 1 1 1
0 0 0 0 1 1
1 1 1 0 0 0
```

Sample input and output

The input will be given in the following format:

```
0 0 0 0 0 0
1 1 1 1 1 0
0 0 0 0 0 0
0 1 1 1 1 1
0 0 0 0 1 1
1 1 1 0 0 0
```

Maze data. We will always work with 6 x 6 arrays

```
0 0
```

start (row, col)

```
5 5
```

target (row, col)

The path should be printed to the *stdout* in the following (row, col) format.

```
 0 (0, 0 )
 1 (0, 1 )
 2 (0, 2 )
 3 (0, 3 )
 4 (0, 4 )
 5 (0, 5 )
 6 (1, 5 )
 7 (2, 5 )
 8 (2, 4 )
 9 (2, 3 )
10 (2, 2 )
11 (2, 1 )
12 (2, 0 )
13 (3, 0 )
14 (4, 0 )
15 (4, 1 )
16 (4, 2 )
17 (4, 3 )
18 (5, 3 )
19 (5, 4 )
20 (5, 5 )
```

```
/* 2-d maze search program */

#include <iostream>
using namespace std;

#define X_DIM 6
#define Y_DIM 6

struct coord_type
{
   int y;
   int x;
}  path [Y_DIM * X_DIM];

int maze[Y_DIM][X_DIM];
int visited[Y_DIM][X_DIM];

int starty, startx;      // (y, x) start coordinates
int targety, targetx;    // (y, x) target coordinates
int truelen;             // length of the final path

void read_maze(
void)
{
   int i = 0;
   int *loc = maze[0];

   while (i < X_DIM * Y_DIM)
   {
      cin >> *loc;
      loc += 1;
      i += 1;
   }
   cin >> starty  >> startx;
   cin >> targety >> targetx;
}
```

Because recursion involves a considerable amount of function call/return activity, overhead can be minimized considerably by making global those variables not required for the recursion to work.

The *legal_move()* function

Since we have to evaluate the possibilty of moving East, South, West, and North, it is best to write a single function that will return *true* if a potential move is legal and *false* if it is not.

```
int legal_move(
int desty,     // potential destination for next step
int destx)
{
```

if ((desty, destx) is

- *in the maze and*
- *hasn't been previously visited*
- *has a value of 0*
- `return(1);`

else

```
  return(0);  // false -> not legal move

}
```

The *build_path()* function

The *build_path()* function is the recursive function that actually solves the problem. The variable *pathlen* carries the current *depth* of the recursion which is also the current length of the *path.*

Because these algorithms are very subtle it is even for even an experience *gdb* user to have trouble keeping track what is going on and going wrong. Thus it is a good idea to put diagnostic prints at all entry and exit points of the recursive routine.

```
int build_path(
int pathlen,     // current length of path
int y,           // (y, x) coordinates of this location
int x)
{
     int rc = -1;
     remember this location has been visited
     fprintf(stderr, "visiting %d %d with pathlen %d \n", y, x,
                       pathlen);
     if this location is the target
     {
       fprintf(stderr, "found the target at %d %d \n", y, x);
       remember current pathlen in truelen
       store this location in the path
        return(0);
     }
```

```
/* Haven't reached the target so need to press onward */
```

try to build_path() east;
if that doesn't work try to build_path() south;
if that doesn't work try to build_path() west;
if that doesn't work try to build_path() north;

If the target was found, then *build_path()* returns a non-negative number and the search is over. *Note that the path is built backward as the recursion unwinds.*

```
    {
       fprintf(stderr, "adding point %d %d with pathlen %d \n",
                                y, x, pathlen);
        store this location in  the path
       return(0);

    }

/* Hit a dead end... back up one spot */

   if nothing worked
   {
     fprintf(stderr, "stuck at %d %d backing up \n", y, x);
     return(-1)
}


int main()
{
   int pathlen = 0;

   read_maze();

   pathlen = build_path(pathlen, starty, startx);
   if (pathlen)
      print_path(truelen);
   else
      printf("can't get there from here! \n");

}
```

```
0 0 0 0 0 0
1 1 0 1 1 1
1 0 0 0 0 0
0 0 1 1 1 1
0 1 0 0 0 1
0 0 0 1 0 0

5 4
0 5

visiting 5 4 with pathlen 0
visiting 5 5 with pathlen 1
stuck at 5 5 backing up
visiting 4 4 with pathlen 1
visiting 4 3 with pathlen 2
visiting 4 2 with pathlen 3
visiting 5 2 with pathlen 4
visiting 5 1 with pathlen 5
visiting 5 0 with pathlen 6
visiting 4 0 with pathlen 7
visiting 3 0 with pathlen 8
visiting 3 1 with pathlen 9
visiting 2 1 with pathlen 10
visiting 2 2 with pathlen 11
visiting 2 3 with pathlen 12
visiting 2 4 with pathlen 13
visiting 2 5 with pathlen 14
stuck at 2 5 backing up
stuck at 2 4 backing up
stuck at 2 3 backing up
visiting 1 2 with pathlen 12
visiting 0 2 with pathlen 13
visiting 0 3 with pathlen 14
visiting 0 4 with pathlen 15
visiting 0 5 with pathlen 16
found the target at 0 5
```

Be sure to disable all diagnostic prints before you submit.

In this lab you will submit a single file, maze.c that includes the new and updated class methods constructed as part of this lab.

```
sendlab.102.labsection#  lab#  maze.cpp
```