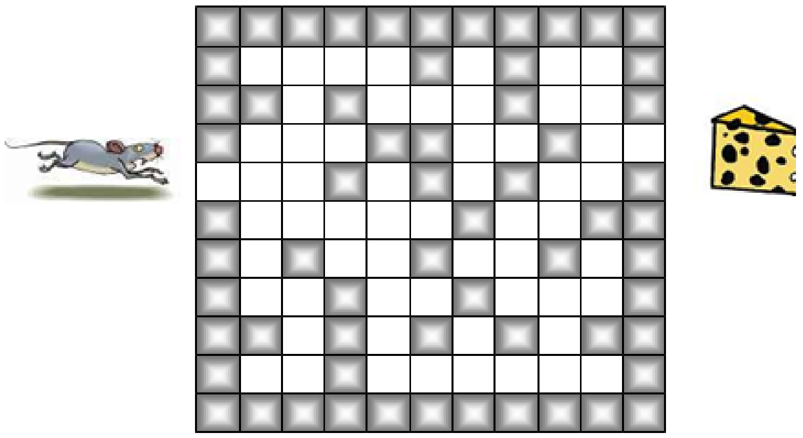
Maze

Topic Coverage

- Assignment and expressions
- Nested control statements
- Functions and procedures
- Character strings

Problem Description

Solving a maze involves finding a route through the maze from start to finish. A typical example of a maze is shown below.

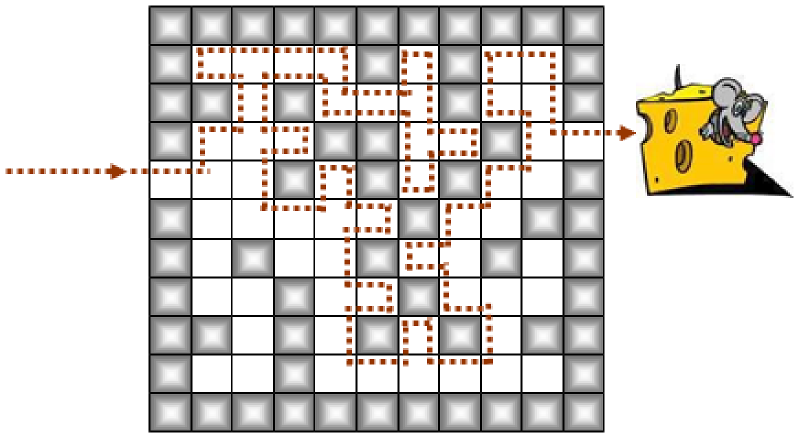


An `n`-by-`n` maze can be represented as a square grid with `@` denoting walls and `.` denoting open positions. Below is an example of a maze of size `11`-by-`11` with the starting and ending positions denoted by `s` and `e` respectively.

```
@@@@@@@@@@@
@....@.@..@
@@.@...@..@
@...@@..@.e
s..@.@.@..@
@.....@..@@
@.@..@..@.@
@..@..@...@
@@.@.@.@.@@
@..@......@
@@@@@@@@@@@
```

By assuming that the top left location is `(0, 0)`, the example above shows $(r_s, c_s) = (4, 0)$ and $(r_e, c_e) = (3, 10)$.

One way to traverse the maze is to follow the **left-hand-rule** as shown below.



Starting at location `(4, 0)` and facing eastwards, hold out your left hand and start walking. The following shows the first seven steps of traversal from `(4, 0)` to `(1, 1)`. Each time a location is stepped on, the counter associated with that position is increased by 1.

```
@@@@@@@@@@@
@11..@.@..@
@@1@...@..@
@11.@@..@.e
11.@.@.@..@
@.....@..@@
@.@..@..@.@
@..@..@...@
@@.@.@.@.@@
@..@......@
@@@@@@@@@@@
```

While keeping the left hand on the wall at location `(1, 1)`, further traversal leads us out of the dead-end into location `(1, 2)` as shown below. Notice that the number of times visited is updated to `2`.

```
@@@@@@@@@@@
@12..@.@..@
@@1@...@..@
@11.@@..@.e
11.@.@.@..@
@.....@..@@
@.@..@..@.@
@..@..@...@
@@.@.@.@.@@
@..@......@
@@@@@@@@@@@
```

Keep following the wall until you reach the end at `(3, 10)`. In the diagram below, numbers along the path denote the number of times the location was visited when traversing the maze from start to end.

```
@@@@@@@@@@@
@1322@1@11@
@@2@223@12@
@131@@31@21
111@1@1@11@
@.1131@11@@
@.@.1@12@.@
@..@21@11.@
@@.@1@1@1@@
@..@11211.@
@@@@@@@@@@@
```

Task

Write a program that reads in a maze of size `n`-by-`n`, traverses the maze using the left-hand-rule and outputs the path taken in terms of the number of visits for each location along the path.

Take note of the following:

- `n` is an integer between `3` and `30` inclusive.
- The start position `s` is located on the left edge (except the two corners).
- The end position `e` can be located at any of the four edges (except the four corners), and not the same as the start position.
- Apart from the characters `s` and `e`, the edges of the maze comprises `@`, each denoting a wall.
- Only one sample test case is provided to test for format correctness. You should device your own test cases to test your program.

This task is divided into several levels. Read through all the levels (from first to last, then from last to first) to see how the different levels are related. **You may start from any level.**

- **Highest assessible level for this lab is level 6; you MUST submit a program for level 6 to gain full credit for this lab.**
- **Level 7 is a bonus level; attempt and submit level 7 after you have submitted the program for level 6.**
- **Deadline: Submit your work to CodeCrunch by Thursday, 10 November, 23:59:59.**

Level 1

Name your program `maze1.c`

Write a program that reads in a maze of size `n`-by-`n` and outputs the maze.

The following is a sample run of the program. User input is underlined. Ensure that the last line of output is followed by a newline character.

```
$ ./a.out
@@@@@@@@@@@
@....@.@..@
@@.@...@..@
@...@@..@.e
s..@.@.@..@
@.....@..@@
@.@..@..@.@
@..@..@...@
@@.@.@.@.@@
@..@......@
@@@@@@@@@@@
@@@@@@@@@@@
@....@.@..@
@@.@...@..@
@...@@..@.e
s..@.@.@..@
@.....@..@@
@.@..@..@.@
@..@..@...@
@@.@.@.@.@@
@..@......@
@@@@@@@@@@@
```

Check the correctness of the output by typing the following Unix command

```
./a.out < maze.in | diff - maze1.out
```

To proceed to the next level (say level 2), copy your program by typing the Unix command

```
cp maze1.c maze2.c
```

Level 2

Name your program `maze2.c`

Write a program that reads in a maze of size `n`-by-`n` and outputs the maze, as well as the start location.

The following is a sample run of the program. User input is underlined. Ensure that the last line of output is followed by a newline character.

```
$ ./a.out
@@@@@@@@@@@
@....@.@..@
@@.@...@..@
@...@@..@.e
s..@.@.@..@
@.....@..@@
```

```
@.@..@..@.@
@..@..@...@
@@.@.@.@.@@
@..@......@
@@@@@@@@@@@
start = (4,0)
@@@@@@@@@@@
@....@.@..@
@@.@...@..@
@...@@..@.e
s..@.@.@..@
@.....@..@@
@.@..@..@.@
@..@..@...@
@@.@.@.@.@@
@..@......@
@@@@@@@@@@@
```

Check the correctness of the output by typing the following Unix command

```
./a.out < maze.in | diff - maze2.out
```

To proceed to the next level (say level 3), copy your program by typing the Unix command

```
cp maze2.c maze3.c
```

Level 3

Name your program `maze3.c`

Write a program that reads in a maze of size `n`-by-`n` and outputs the maze, as well as the start location. With the default direction facing eastwards, move forward until a wall or the end location is reached. Output the maze again showing the number of times the location is stepped upon.

The following is a sample run of the program. User input is underlined. Ensure that the last line of output is followed by a newline character.

```
$ ./a.out
@@@
s.e
@@@
start = (1,0)
@@@
111
@@@
```

```
$ ./a.out
@@@@@@@@@@@
@....@.@..@
@@.@...@..@
@...@@..@.e
s..@.@.@..@
@.....@..@@
@.@..@..@.@
@..@..@...@
@@.@.@.@.@@
@..@......@
@@@@@@@@@@@
start = (4,0)
@@@@@@@@@@@
@....@.@..@
@@.@...@..@
@...@@..@.e
111@.@.@..@
@.....@..@@
@.@..@..@.@
```

```
@..@..@...@
@@.@.@.@.@@
@..@......@
@@@@@@@@@@@
```

Check the correctness of the output by typing the following Unix command

```
./a.out < maze.in | diff - maze3.out
```

To proceed to the next level (say level 4), copy your program by typing the Unix command

```
cp maze3.c maze4.c
```

Level 4

Name your program `maze4.c`

Write a program that reads in a maze of size `n`-by-`n` and outputs the maze, as well as the start location. With the initial direction facing eastwards, traverse the maze using the following algorithm.

```
if (there is an open position on the left) {
   turn left;
}
move one step;
```

Keep traversing the maze until the the end location is reached, or no more left turns can be made. Output the maze again showing the number of times the location is stepped upon.

The following is a sample run of the program. User input is underlined. Ensure that the last line of output is followed by a newline character.

```
$ ./a.out
@@@
s.e
@@@
start = (1,0)
@@@
111
@@@
```

```
$ ./a.out
@@@e@
@@@.@
@@@.@
s...@
@@@@@
start = (3,0)
@@@1@
@@@1@
@@@1@
1111@
@@@@@
```

```
$ ./a.out
@@@@@
e...@
@@@.@
s...@
@@@@@
start = (3,0)
@@@@@
1111@
@@@1@
1111@
@@@@@
```

```
$ ./a.out
```

```
@@@@@
@...@
@.@.@
s...@
@e@@@
start = (3,0)
@@@@@
@1..@
@1@.@
11..@
@e@@@
```

```
$ ./a.out
@@@@@@@@@@@
@....@.@..@
@@.@...@..@
@...@@..@.e
s..@.@.@..@
@.....@..@@
@.@..@..@.@
@..@..@...@
@@.@.@.@.@@
@..@......@
@@@@@@@@@@@
start = (4,0)
@@@@@@@@@@@
@....@.@..@
@@.@...@..@
@1..@@..@.e
11.@.@.@..@
@.....@..@@
@.@..@..@.@
@..@..@...@
@@.@.@.@.@@
@..@......@
@@@@@@@@@@@
```

Check the correctness of the output by typing the following Unix command

```
./a.out < maze.in | diff - maze4.out
```

To proceed to the next level (say level 5), copy your program by typing the Unix command

```
cp maze4.c maze5.c
```

Level 5

Name your program `maze5.c`

Write a program that reads in a maze of size `n`-by-`n` and outputs the maze, as well as the start location. With the initial direction facing eastwards, traverse the maze using the following algorithm.

```
if (there is an open position on the left) {
   turn left;
} else if (there is an open position in front) {
   stay put;
} else if (there is an open position in right) {
   turn right;
}
move one step;
```

Keep traversing until a wall, the end location or a location that has been visited previously is reached. Output the maze again showing the number of times the location is stepped upon.

The following is a sample run of the program. User input is underlined. Ensure that the last line of output is followed by a newline character.

```
$ ./a.out
@@@
s.e
@@@
start = (1,0)
@@@
111
@@@
```

```
$ ./a.out
@@@e@
@@@.@
@@@.@
s...@
@@@@@
start = (3,0)
@@@1@
@@@1@
@@@1@
1111@
@@@@@
```

```
$ ./a.out
@@@@@
e...@
@@@.@
s...@
@@@@@
start = (3,0)
@@@@@
1111@
@@@1@
1111@
@@@@@
```

```
$ ./a.out
@@@@@
@...@
@.@.@
s...@
@e@@@
start = (3,0)
@@@@@
@111@
@1@1@
1111@
@e@@@
```

```
$ ./a.out
@@@@@@@@@@@
@....@.@..@
@@.@...@..@
@...@@..@.e
s..@.@.@..@
@.....@..@@
@.@..@..@.@
@..@..@...@
@@.@.@.@.@@
@..@......@
@@@@@@@@@@@
start = (4,0)
@@@@@@@@@@@
@11..@.@..@
@@1@...@..@
@11.@@..@.e
11.@.@.@..@
@.....@..@@
@.@..@..@.@
@..@..@...@
@@.@.@.@.@@
@..@......@
@@@@@@@@@@@
```

Check the correctness of the output by typing the following Unix command

```
./a.out < maze.in | diff - maze5.out
```

To proceed to the next level (say level 6), copy your program by typing the Unix command

```
cp maze5.c maze6.c
```

Level 6

Name your program `maze6.c`

Write a program that reads in a maze of size `n`-by-`n` and outputs the maze, as well as the start location. With the initial direction facing eastwards, traverse the maze using the following algorithm.

```
if (there is an open position on the left) {
   turn left;
} else if (there is an open position in front) {
   stay put;
} else if (there is an open position in right) {
   turn right;
} else {
   turn back;
}
move one step;
```

Keep traversing until the end location is reached. Output the maze again showing the number of times the location is stepped upon.

The following is a sample run of the program. User input is underlined. Ensure that the last line of output is followed by a newline character.

```
$ ./a.out
@@@@@
@...@
@.@.@
s...@
@e@@@
start = (3,0)
@@@@@
@111@
@1@1@
1211@
@1@@@
```

```
$ ./a.out
@@@@@@@@@@@
@....@.@..@
@@.@...@..@
@...@@..@.e
s..@.@.@..@
@.....@..@@
@.@..@..@.@
@..@..@...@
@@.@.@.@.@@
@..@......@
@@@@@@@@@@@
start = (4,0)
@@@@@@@@@@@
@1322@1@11@
@@2@223@12@
@131@@31@21
111@1@1@11@
@.1131@11@@
@.@.1@12@.@
@..@21@11.@
```

```
@@.@1@1@1@@
@..@11211.@
@@@@@@@@@@@
```

Check the correctness of the output by typing the following Unix command

```
./a.out < maze.in | diff - maze6.out
```

To proceed to the next level (say level 7), copy your program by typing the Unix command

```
cp maze6.c maze7.c
```

Level 7 (Bonus)

Name your program `maze7.c`

Write a program that reads in a maze of size `n`-by-`n` and outputs the maze, as well as the start location. With the initial direction facing eastwards, traverse the maze using the following algorithm.

```
if (there is an open position on the left) {
   turn left;
} else if (there is an open position in front) {
   stay put;
} else if (there is an open position in right) {
   turn right;
} else {
   turn back;
}
move one step;
```

Keep traversing until the end location is reached. Output the maze again showing the number of times the location is stepped upon.

Using the path found by the **left-hand-rule** above, determining the shortest possible path and output this path using asterisks `*`.

The following is a sample run of the program. User input is underlined. Ensure that the last line of output is followed by a newline character.

```
$ ./a.out
@@@@@@@@@@@
@....@.@..@
@@.@...@..@
@...@@..@.e
s..@.@.@..@
@.....@..@@
@.@..@..@.@
@..@..@...@
@@.@.@.@.@@
@..@......@
@@@@@@@@@@@
start = (4,0)
@@@@@@@@@@@
@1322@1@11@
@@2@223@12@
@131@@31@21
111@1@1@11@
@.1131@11@@
@.@.1@12@.@
@..@21@11.@
@@.@1@1@1@@
@..@11211.@
@@@@@@@@@@@
Shortest path:
@@@@@@@@@@@
```

```
@....@.@..@
@@.@...@..@
@...@@..@**
***@.@.@**@
@.***.@**@@
@.@.*@.*@.@
@..@*.@**.@
@@.@*@.@*@@
@..@*****.@
@@@@@@@@@@@
```

Check the correctness of the output by typing the following Unix command

```
./a.out < maze.in | diff - maze7.out
```