PR3 Maze Explanation

Mark Redekopp

Maze Solver

- Consider this maze
  - S = Start
  - F = Finish
  - . = Free
  - # = Wall
- Find the shortest path

(0,0) .	(0,1) .	(0,2) .	(0,3) .
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0) .	(2,1) #	(2,2) .	(2,3) #
(3,0) .	(3,1) .	(3,2) .	(3,3) #

Maze Solver

- To find ***a*** shortest path (since there might be many), we can use the breadth-first search (BFS) algorithm
- Since we can only visit "1 location at a time" we must keep track of (store) the locations we want to visit until we can actually process them
- BFS requires we visit all nearer squares before further squares
  - A simple way to meet this requirement is to make a square "get in line" (i.e. a queue) when we encounter it
  - We will pull squares from the front to explore and add new squares to the back of the queue

Maze array:

(0,0) .	(0,1) .	(0,2) .	(0,3) .
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0) .	(2,1) #	(2,2) .	(2,3) #
(3,0) .	(3,1) .	(3,2) .	(3,3) #

Queue

Maze Solver

- We begin the algorithm by putting the starting location into the queue

Maze array:

(0,0) .	(0,1) .	(0,2) .	(0,3) .
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0) .	(2,1) #	(2,2) .	(2,3) #
(3,0) .	(3,1) .	(3,2) .	(3,3) #

Queue

1,0															

Maze Solver

- We begin by putting the starting location into the queue
- Then we enter a loop…while the queue is not empty
  - Extract the front location, call it "curr"
  - Visit each neighbor (N,W,S,E) one at a time
  - If the neighbor is the finish
    - Stop and trace backwards
  - Else if the neighbor is a valid location and not visited before
    - Then add it to the back of the queue
    - Mark it as visited so we don't add it to the queue again
    - Record its "predecessor" (the location [i.e. curr] that found this neighbor

Maze array:

(0,0) .	(0,1) .	(0,2) .	(0,3) .
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0) .	(2,1) #	(2,2) .	(2,3) #
(3,0) .	(3,1) .	(3,2) .	(3,3) #

Queue

1,0															

Visited

(0,0) 0	(0,1) 0	(0,2) 0	(0,3) 0
(1,0) 1	(1,1) 0	(1,2) 0	(1,3) 0
(2,0) 0	(2,1) 0	(2,2) 0	(2,3) 0
(3,0) 0	(3,1) 0	(3,2) 0	(3,3) 0

Predecessor

(0,0) -1,-1	(0,1) -1,-1	(0,2) -1,-1	(0,3) -1,-1
(1,0) -1,-1	(1,1) -1,-1	(1,2) -1,-1	(1,3) -1,-1
(2,0) -1,-1	(2,1) -1,-1	(2,2) -1,-1	(2,3) -1,-1
(3,0) -1,-1	(3,1) -1,-1	(3,2) -1,-1	(3,3) -1,-1

Maze Solver

- We begin by putting the starting location into the queue
- Then we enter a loop…while the queue is not empty
  - Extract the front location, call it "curr"
  - Visit each neighbor (N,W,S,E) one at a time
  - If the neighbor is the finish
    - Stop and trace backwards
  - Else if the neighbor is a valid location and not visited before
    - Then add it to the back of the queue
    - Mark it as visited so we don't add it to the queue again
    - Record its predecessor (the location [i.e. curr] that found this neighbor

Maze array:

(0,0) .	(0,1) .	(0,2) .	(0,3) .
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0) .	(2,1) #	(2,2) .	(2,3) #
(3,0) .	(3,1) .	(3,2) .	(3,3) #

curr = 1,0

Queue

1,0	0,0	2,0													

Visited

(0,0) 1	(0,1) 0	(0,2) 0	(0,3) 0
(1,0) 1	(1,1) 0	(1,2) 0	(1,3) 0
(2,0) 1	(2,1) 0	(2,2) 0	(2,3) 0
(3,0) 0	(3,1) 0	(3,2) 0	(3,3) 0

Predecessor

(0,0) 1,0	(0,1) -1,-1	(0,2) -1,-1	(0,3) -1,-1
(1,0) -1,-1	(1,1) -1,-1	(1,2) -1,-1	(1,3) -1,-1
(2,0) 1,0	(2,1) -1,-1	(2,2) -1,-1	(2,3) -1,-1
(3,0) -1,-1	(3,1) -1,-1	(3,2) -1,-1	(3,3) -1,-1

Maze Solver

- We begin by putting the starting location into the queue
- Then we enter a loop…while the queue is not empty
  - Extract the front location, call it "curr"
  - Visit each neighbor (N,W,S,E) one at a time
  - If the neighbor is the finish
    - Stop and trace backwards
  - Else if the neighbor is a valid location and not visited before
    - Then add it to the back of the queue
    - Mark it as visited so we don't add it to the queue again
    - Record its predecessor (the location [i.e. curr] that found this neighbor

Maze array:

(0,0) .	(0,1) .	(0,2) .	(0,3) .
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0) .	(2,1) #	(2,2) .	(2,3) #
(3,0) .	(3,1) .	(3,2) .	(3,3) #

curr = 0,0

Queue

1,0	0,0	2,0	0,1												

Visited

(0,0) 1	(0,1) 1	(0,2) 0	(0,3) 0
(1,0) 1	(1,1) 0	(1,2) 0	(1,3) 0
(2,0) 1	(2,1) 0	(2,2) 0	(2,3) 0
(3,0) 0	(3,1) 0	(3,2) 0	(3,3) 0

Predecessor

(0,0) 1,0	(0,1) 0,0	(0,2) -1,-1	(0,3) -1,-1
(1,0) -1,-1	(1,1) -1,-1	(1,2) -1,-1	(1,3) -1,-1
(2,0) 1,0	(2,1) -1,-1	(2,2) -1,-1	(2,3) -1,-1
(3,0) -1,-1	(3,1) -1,-1	(3,2) -1,-1	(3,3) -1,-1

Maze Solver

- We begin by putting the starting location into the queue
- Then we enter a loop…while the queue is not empty
  - Extract the front location, call it "curr"
  - Visit each neighbor (N,W,S,E) one at a time
  - If the neighbor is the finish
    - Stop and trace backwards
  - Else if the neighbor is a valid location and not visited before
    - Then add it to the back of the queue
    - Mark it as visited so we don't add it to the queue again
    - Record its predecessor (the location [i.e. curr] that found this neighbor

Maze array:

(0,0) .	(0,1) .	(0,2) .	(0,3) .
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0) .	(2,1) #	(2,2) .	(2,3) #
(3,0) .	(3,1) .	(3,2) .	(3,3) #

curr = 2,0

Queue

1,0	0,0	2,0	0,1	3,0											

Visited

(0,0) 1	(0,1) 1	(0,2) 0	(0,3) 0
(1,0) 1	(1,1) 0	(1,2) 0	(1,3) 0
(2,0) 1	(2,1) 0	(2,2) 0	(2,3) 0
(3,0) 1	(3,1) 0	(3,2) 0	(3,3) 0

Predecessor

(0,0) 1,0	(0,1) 0,0	(0,2) -1,-1	(0,3) -1,-1
(1,0) -1,-1	(1,1) -1,-1	(1,2) -1,-1	(1,3) -1,-1
(2,0) 1,0	(2,1) -1,-1	(2,2) -1,-1	(2,3) -1,-1
(3,0) 2,0	(3,1) -1,-1	(3,2) -1,-1	(3,3) -1,-1

Maze Solver

- We begin by putting the starting location into the queue
- Then we enter a loop…while the queue is not empty
  - Extract the front location, call it "curr"
  - Visit each neighbor (N,W,S,E) one at a time
  - If the neighbor is the finish
    - Stop and trace backwards
  - Else if the neighbor is a valid location and not visited before
    - Then add it to the back of the queue
    - Mark it as visited so we don't add it to the queue again
    - Record its predecessor (the location [i.e. curr] that found this neighbor

Maze array:

(0,0) .	(0,1) .	(0,2) .	(0,3) .
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0) .	(2,1) #	(2,2) .	(2,3) #
(3,0) .	(3,1) .	(3,2) .	(3,3) #

curr = 0,1

Queue

1,0	0,0	2,0	0,1	3,0	0,2										

Visited

(0,0) 1	(0,1) 1	(0,2) 1	(0,3) 0
(1,0) 1	(1,1) 0	(1,2) 0	(1,3) 0
(2,0) 1	(2,1) 0	(2,2) 0	(2,3) 0
(3,0) 1	(3,1) 0	(3,2) 0	(3,3) 0

Predecessor

(0,0) 1,0	(0,1) 0,0	(0,2) 0,1	(0,3) -1,-1
(1,0) -1,-1	(1,1) -1,-1	(1,2) -1,-1	(1,3) -1,-1
(2,0) 1,0	(2,1) -1,-1	(2,2) -1,-1	(2,3) -1,-1
(3,0) 2,0	(3,1) -1,-1	(3,2) -1,-1	(3,3) -1,-1

Maze Solver

- We begin by putting the starting location into the queue
- Then we enter a loop…while the queue is not empty
  - Extract the front location, call it "curr"
  - Visit each neighbor (N,W,S,E) one at a time
  - If the neighbor is the finish
    - Stop and trace backwards
  - Else if the neighbor is a valid location and not visited before
    - Then add it to the back of the queue
    - Mark it as visited so we don't add it to the queue again
    - Record its predecessor (the location [i.e. curr] that found this neighbor

Maze array:

(0,0) .	(0,1) .	(0,2) .	(0,3) .
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0) .	(2,1) #	(2,2) .	(2,3) #
(3,0) .	(3,1) .	(3,2) .	(3,3) #

curr = 3,0

Queue

1,0	0,0	2,0	0,1	3,0	0,2	3,1									

Visited

(0,0) 1	(0,1) 1	(0,2) 1	(0,3) 0
(1,0) 1	(1,1) 0	(1,2) 0	(1,3) 0
(2,0) 1	(2,1) 0	(2,2) 0	(2,3) 0
(3,0) 1	(3,1) 1	(3,2) 0	(3,3) 0

Predecessor

(0,0) 1,0	(0,1) 0,0	(0,2) 0,1	(0,3) -1,-1
(1,0) -1,-1	(1,1) -1,-1	(1,2) -1,-1	(1,3) -1,-1
(2,0) 1,0	(2,1) -1,-1	(2,2) -1,-1	(2,3) -1,-1
(3,0) 2,0	(3,1) 3,0	(3,2) -1,-1	(3,3) -1,-1

Maze Solver

- We begin by putting the starting location into the queue
- Then we enter a loop…while the queue is not empty
  - Extract the front location, call it "curr"
  - Visit each neighbor (N,W,S,E) one at a time
  - If the neighbor is the finish
    - Stop and trace backwards
  - Else if the neighbor is a valid location and not visited before
    - Then add it to the back of the queue
    - Mark it as visited so we don't add it to the queue again
    - Record its predecessor (the location [i.e. curr] that found this neighbor

Maze array:

(0,0) .	(0,1) .	(0,2) .	(0,3) .
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0) .	(2,1) #	(2,2) .	(2,3) #
(3,0) .	(3,1) .	(3,2) .	(3,3) #

curr = 0,2

Found the Finish at (1,2)

Queue

1,0	0,0	2,0	0,1	3,0	0,2	3,1									

Visited

(0,0) 1	(0,1) 1	(0,2) 1	(0,3) 0
(1,0) 1	(1,1) 0	(1,2) 0	(1,3) 0
(2,0) 1	(2,1) 0	(2,2) 0	(2,3) 0
(3,0) 1	(3,1) 1	(3,2) 0	(3,3) 0

Predecessor

(0,0) 1,0	(0,1) 0,0	(0,2) 0,1	(0,3) -1,-1
(1,0) -1,-1	(1,1) -1,-1	(1,2) -1,-1	(1,3) -1,-1
(2,0) 1,0	(2,1) -1,-1	(2,2) -1,-1	(2,3) -1,-1
(3,0) 2,0	(3,1) 3,0	(3,2) -1,-1	(3,3) -1,-1

Maze Solver

- Now we need to backtrack and add *'s to our shortest path
- We use the predecessor array to walk backwards from curr to the start
  - Set `maze[curr] = '*'`
    - Not real syntax (as ‘curr’ is a Location struct)
  - Update current to its predecessor

Maze array:

(0,0) .	(0,1) .	(0,2) *	(0,3) .
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0) .	(2,1) #	(2,2) .	(2,3) #
(3,0) .	(3,1) .	(3,2) .	(3,3) #

curr = 0,2

curr = <predecessor>

Queue

1,0	0,0	2,0	0,1	3,0	0,2	3,1									

Visited

(0,0) 1	(0,1) 1	(0,2) 1	(0,3) 0
(1,0) 1	(1,1) 0	(1,2) 0	(1,3) 0
(2,0) 1	(2,1) 0	(2,2) 0	(2,3) 0
(3,0) 1	(3,1) 1	(3,2) 0	(3,3) 0

Predecessor

(0,0) 1,0	(0,1) 0,0	(0,2) 0,1	(0,3) -1,-1
(1,0) -1,-1	(1,1) -1,-1	(1,2) -1,-1	(1,3) -1,-1
(2,0) 1,0	(2,1) -1,-1	(2,2) -1,-1	(2,3) -1,-1
(3,0) 2,0	(3,1) 3,0	(3,2) -1,-1	(3,3) -1,-1

Maze Solver

- Now we need to backtrack and add *'s to our shortest path
- We use the predecessor array to walk backwards from curr to the start
  - Set `maze[curr] = '*'`
    - Not real syntax (as ‘curr’ is a Location struct)
  - Update current to its predecessor

Maze array:

(0,0) .	(0,1) *	(0,2) *	(0,3) .
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0) .	(2,1) #	(2,2) .	(2,3) #
(3,0) .	(3,1) .	(3,2) .	(3,3) #

curr = 0,2

curr = <predecessor>

Queue

1,0	0,0	2,0	0,1	3,0	0,2	3,1									

Visited

(0,0) 1	(0,1) 1	(0,2) 1	(0,3) 0
(1,0) 1	(1,1) 0	(1,2) 0	(1,3) 0
(2,0) 1	(2,1) 0	(2,2) 0	(2,3) 0
(3,0) 1	(3,1) 1	(3,2) 0	(3,3) 0

Predecessor

(0,0) 1,0	(0,1) 0,0	(0,2) 0,1	(0,3) -1,-1
(1,0) -1,-1	(1,1) -1,-1	(1,2) -1,-1	(1,3) -1,-1
(2,0) 1,0	(2,1) -1,-1	(2,2) -1,-1	(2,3) -1,-1
(3,0) 2,0	(3,1) 3,0	(3,2) -1,-1	(3,3) -1,-1

Maze Solver

- Now we need to backtrack and add *'s to our shortest path
- We use the predecessor array to walk backwards from curr to the start
  - Set `maze[curr] = '*'`
    - Not real syntax (as ‘curr’ is a Location struct)
  - Update current to its predecessor

Maze array:

(0,0) *	(0,1) *	(0,2) *	(0,3) .
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0) .	(2,1) #	(2,2) .	(2,3) #
(3,0) .	(3,1) .	(3,2) .	(3,3) #

curr = 0,2

curr = <predecessor>

Queue

1,0	0,0	2,0	0,1	3,0	0,2	3,1									

Visited

(0,0) 1	(0,1) 1	(0,2) 1	(0,3) 0
(1,0) 1	(1,1) 0	(1,2) 0	(1,3) 0
(2,0) 1	(2,1) 0	(2,2) 0	(2,3) 0
(3,0) 1	(3,1) 1	(3,2) 0	(3,3) 0

Predecessor

(0,0) 1,0	(0,1) 0,0	(0,2) 0,1	(0,3) -1,-1
(1,0) -1,-1	(1,1) -1,-1	(1,2) -1,-1	(1,3) -1,-1
(2,0) 1,0	(2,1) -1,-1	(2,2) -1,-1	(2,3) -1,-1
(3,0) 2,0	(3,1) 3,0	(3,2) -1,-1	(3,3) -1,-1

Need to Do

- Queue class
  - Make internal array to be of size = max number of squares
  - Should it be dynamic?
  - We need to keep track of the "front" and "back" since only a portion of the array is used
  - Just use integer indexes to record where the front and back are

Maze array:

(0,0) *	(0,1) *	(0,2) *	(0,3) .
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0) .	(2,1) #	(2,2) .	(2,3) #
(3,0) .	(3,1) .	(3,2) .	(3,3) #

curr = 2,0

Queue

1,0	0,0	2,0	0,1	3,0											

front 3

back 5

Visited

(0,0) 1	(0,1) 1	(0,2) 1	(0,3) 0
(1,0) 1	(1,1) 0	(1,2) 0	(1,3) 0
(2,0) 1	(2,1) 0	(2,2) 0	(2,3) 0
(3,0) 1	(3,1) 1	(3,2) 0	(3,3) 0

Predecessor

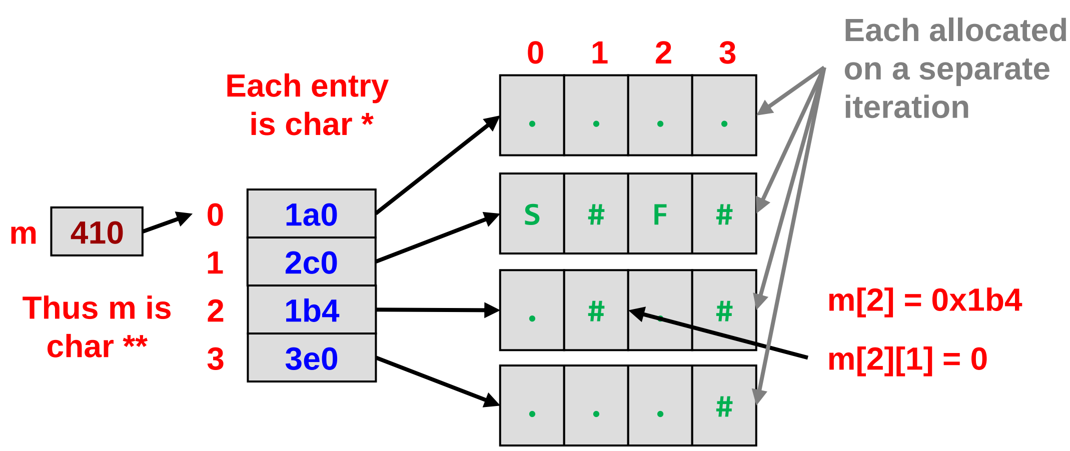
(0,0) 1,0	(0,1) 0,0	(0,2) 0,1	(0,3) -1,-1
(1,0) -1,-1	(1,1) -1,-1	(1,2) -1,-1	(1,3) -1,-1
(2,0) 1,0	(2,1) -1,-1	(2,2) -1,-1	(2,3) -1,-1
(3,0) 2,0	(3,1) 3,0	(3,2) -1,-1	(3,3) -1,-1

Need to Do

- Allocate 2D arrays for maze, visited, and predecessors
  - Note: in C/C++ you cannot allocate a 2D array with variable size dimensions
    - **BAD: new char[numrows][numcols];**
  - Solution:
    - Allocate 1 array of NUMROWS pointers
    - Then loop through that array and allocate an array of NUMCOLS items and put its start address into the i-th pointer in the array you allocated above

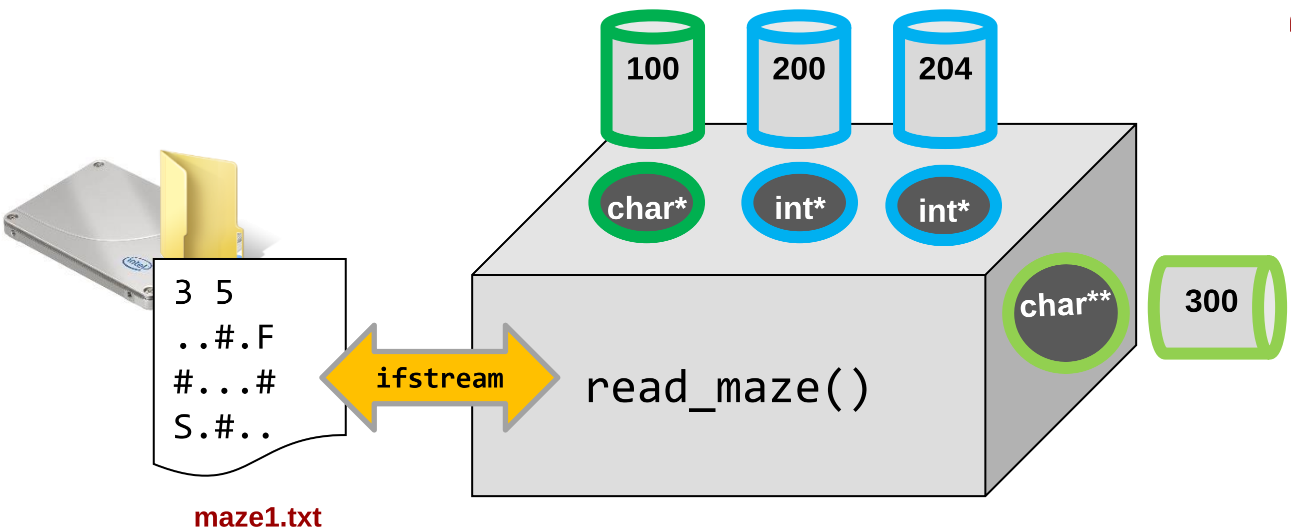
Maze array:

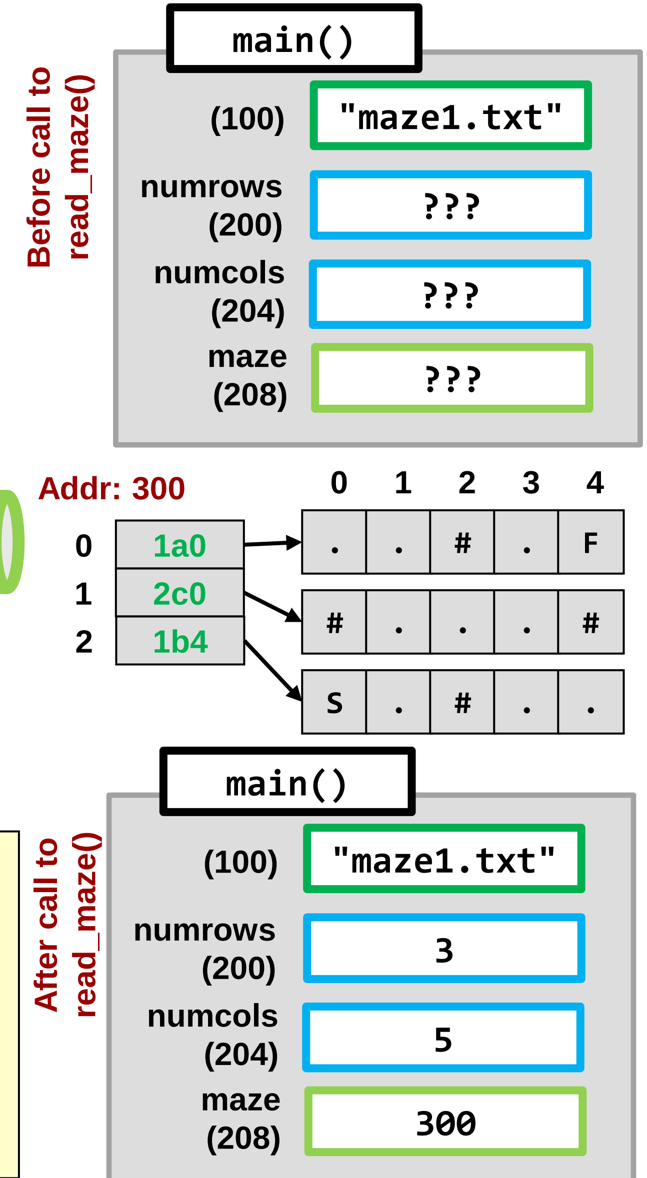
(0,0) .	(0,1) .	(0,2) .	(0,3) .
(1,0) S	(1,1) #	(1,2) F	(1,3) #
(2,0) .	(2,1) #	(2,2) .	(2,3) #
(3,0) .	(3,1) .	(3,2) .	(3,3) #



Code Organization (maze_io.h/cpp)

- Illustration of how read_maze() should work






```
// read maze from the given filename, allocate and
// return maze 2D array as well as filling in rows and cols
char** read_maze(char* filename, int* rows, int* cols);

// print maze to cout
void print_maze(char** maze, int rows, int cols);
```