CS103 Spring 2016: Introduction to Programming

A MAZEING BFS

The world's largest corn maze, in Dixon, California.

In this programming assignment you will write a program to read a given maze (provided as text "art") and find the shortest path from start to finish. Mazes go back to antiquity and the story of the minotaur. However, Theseus didn't have Google Maps.

You will implement *breadth-first search*, a simple algorithm that finds not just any path to the exit in a maze, but actually finds the *shortest* path from the start to the finish. This algorithm was invented in the late 1950s, making it one of the earliest nontrivial algorithms to follow the creation of the electronic computer.

In this assignment you will practice the following skills:

- Dynamic memory allocation of arrays and 2D arrays.
- Using a `struct` and a `class`.
- Finishing an implementation of a queue (queue).
- Implementing the Breadth-First-Search algorithm.

Important Note: Read the entire description all the way through to get an overview. Then go back and re-read it again to start to pull out the tasks you'll need to work on and the order.

Maze Input Format

Your program will receive input from standard input (`cin` , either typed in or redirected from a file). The first line should contain two integer numbers indicating the row and column size of the maze. The number of rows indicated will determine how many lines of text follow (1 row per line). On each line will be one character for each of the indicated number of columns followed by a newline character. The characters can be a period `.` indicating a space in the maze, a `#` sign indicating a wall in the maze, an `S` indicating the start location for your search, or an `F` for the desired finish location. **You can't go outside the grid.** (I.e., you may imagine that lava surrounds the maze perimeter.)

Here is a sample input maze:

```
$ more maze1.in
```

```
4 4
..#.
..#S
F.#.
....
```

In general, it would be

```
rows cols
<row #1: cols many characters>
<row #2: cols many characters>
...
<row #rows: cols many characters>
```

with the possible characters being

Character	Meaning
`.`	Empty space in the maze
`#`	Wall in the maze
`S`	Start location in the maze
`F`	Finish location in the maze

Your search algorithm will find a *shortest* path from the start cell to the finish. It will indicate this path by filling in the character locations on the path with asterisks `*`; then, it will write the resulting character maze to standard output.

Here is a correct output for the input given above:

```
$ ./maze < maze1.in
4 4
..#.
..#S
F*#*
.***
```

Sometimes, no path exists. In this case your program will just output

```
No path could be found!
```

to the screen instead. Also, if the user enters an invalid maze (not 1 start cell or not 1 finish cell), print

```
Invalid maze.
```

Breadth-First Search (BFS)

Breadth First Search is a general technique with many uses including flood fill, shortest paths, and meet-in-the-middle search. The idea is to explore every possible valid location, beginning at the start location, using an ordering so that we always *explore ALL locations at a shorter distance from the start before exploring any location at a longer distance from the start*. In other words, first we "explore" the start, then all locations at distance 1 from the start, then all locations at distance 2 from the start, etc, in that order. This property ensures that when we do find the finish cell, we've arrived there via a shortest-length path. As we search we mark cells that we've explored so that we don't explore them again and so the search doesn't run forever.

How do we ensure the BFS property (italicized above)? We keep a queue of locations in the maze. A queue has the property that the first item added is the first to be removed (first-in, first-out, a.k.a. FIFO). We can easily implement this by always adding new values to the end of a list (i.e. tail of the queue) but remove from the front (i.e. head of the queue). A queue mimics a line of people waiting. New arrivals enter at the back and leave from the front when they are served. The longest/oldest item will be at the front of the queue and the newest at the back.

In our maze search the queue is initially empty and we begin by putting the start location into it. In each iteration, we remove the oldest remaining location from the queue, and we add *all* of its **new** neighbors to the end of the list. This simple algorithm successfully implements the BFS. (If you want the proof, hover your cursor over the paragraph below.)

Each neighbor of a location at distance i from the start must be at distance i-1, i, or i+1 from the start. However, any *new* neighbor must be at distance i+1, since those at i-1 or i were already listed. And every location at distance i+1 has some neighbor at distance i. This is enough to finish the proof.

We keep the BFS going as long as possible, until the queue is exhausted. When this happens, either we reached the finish at some point (and we got there by a shortest path), or we didn't (and no path exists). It only remains to actually mark out the shortest path in `*` characters; this is done by keeping track of *predecessors* as is explained below.

Structs and Classes

The maze in this assignment is a 2D grid. Therefore, each location in the maze is identified by two numbers: a row position and a column position. The approach that we will use for solving the maze will involve keeping track of a list of these locations. It will be extremely convenient to be able to talk about a combined structure that contains both a number for the row and a number for the column; we can accomplish this by creating a simple new data type:

```
struct Location { // define a new data type
   int row;
   int col;
};
```

A `Location` object is simply a package with two numbers that can be manipulated using a period `.` (the member operator); here is an artificial example.

```
Location start;
start.row = 3;
start.col = 5;
Location one_below = start; // make a copy
one_below.row += 1;
cout << start.row << " " << start.col << endl; // 3 5
cout << one_below.row << " " << one_below.col << endl; // 4
```

This will be convenient because we can have an array (or queue) of `Location`s, have functions that take a `Location` as input or output, etc.

Moreover, you will be required to complete the definition of another new data type, the `Queue` class that will be useful in implementing the search. The `Queue` class will store the list of `Locations` waiting to be searched. The `Queue` class should support the following operations:

- create an empty Queue (but with a given maximum capacity of how many Locations it can store)
- add a new `Location` to the back of the Queue
- Remove the oldest `Location` from the front of the Queue
- check if the Queue is empty

When you are writing your higher level BFS algorithm, having this class should make your life easier. Everytime you find a new, unexplored `Location` you will call the Queue's `add_to_back(Location)` function. Every time you want to get the next location to explore from you will call the Queue's `remove_from_front()` function. Internally the `Queue` class should just create an array to hold the maximum number of Locations that could ever be entered into the queue and use integer index variables to remember where it should place new values (i.e. where the back is located) and where it should remove old values (i.e. where the front is located).

Queue Implementation

Remember that our `Queue` class should support the following operations:

Here is an example of how it should behave.

```
// create some locations;
Location three_one, two_two;
three_one.row = 3; three_one.col = 1;
two_two.row = 2; two_two.col = 2;
```

```
// create an queue with max capacity 5
Queue q(5);

cout << boolalpha;
cout << q.is_empty() << endl;            // true
q.add_to_back(three_one);
cout << q.is_empty() << endl;            // false
q.add_to_back(two_two);

Location loc = q.remove_from_front();
cout << loc.row << " " << loc.col << endl; // 3 1
loc = q.remove_from_front();
cout << loc.row << " " << loc.col << endl; // 2 2
cout << q.is_empty() << endl;            // true
```

As you can see, the queue gave us back the *oldest* location (the one `add_to_back` added earliest) first.

We will give you an implementation of this class that is almost complete. It is based on the idea of using a long array/pointer called `contents` that holds `Location` s, as well as two counters:

- `tail` counts the number of `add_to_back` calls so far. Equivalently, its value is the next unused index in the `contents` array.
- `head` counts the number of `remove_from_front` calls so far. Equivalently, its value is the oldest index that has not yet been extracted.

For instance, this is what the internal variables of the `Queue` should look like for the example above, *after both locations are added*:

```
tail: 2
head: 0
contents[0]: (3, 1)
contents[1]: (2, 2)
contents[2..4]: garbage
```

After that, when we make the first call to `remove_from_front`, it should both increase `head` to 1 and return the `Location` (3, 1).

After that, when we make the second call to `remove_from_front`, it should increase `head` to 2 and return the `Location` (2, 2).

After that, because `head` and `tail` are now equal, the `Queue` knows it is empty.

Note: when you delete from the queue, *you do NOT move all the other items*. You simply move the head counter forwards (leaving the old stuff sitting in its original location). Actually, the expense of moving items would make your program much slower. Later in the course you'll learn how to optimize this.

Here are some additional notes about the `Queue` class.

- the `Queue` constructor takes an integer `max_size`. For our BFS application, you should pass `rows*cols` as this maximum size, since that it the maximum number of locations that our `Queue` could be used to explore.
- the constructor syntax is `Queue q(5);` similar to how a file stream is created
- C++ calls the destructor automatically. So while you need to write the code for the destructor you do not call it yourself. We should not see the code `~Queue()` anywhere in `maze.cpp`.

2D Array Allocation

You will not know the size of the maze until runtime, when you read the maze data. Thus we will need to dynamically allocate an array to hold the maze data. Remember that a single call to `new` can only allocate a 1D array. You will need to allocate some 2D arrays in this assignment: one for the maze data, one for the predecessors, and one to remember which grid cells have been visited.

The way to allocate a 2D array in C++ is: use `new[]` once to allocate a 1D array of pointers, then using a loop containing `new[]` to allocate many 1D arrays whose locations are stored in the array of pointers. See the dynamic memory exercise `nxmboard` and `deepnames` in this folder.

Remember that you need to deallocate everything that you allocate. This means that every call to `new[]` needs a matching call to `delete[]`. Otherwise your program will have a memory leak.

Required Functions

In order to create some modularity, your program will be written in several files and should utilize several functions. We will give you skeletons for them.

- `maze.cpp`, the main maze-solving program
- `mazeio.cpp` / `mazeio.h`, routines for input and output
- `queue.cpp` / `queue.h`, the definition of the `Location` struct and `Queue` classes

mazeio.cpp

Your `mazeio.cpp` must define the following functions:

```
char** read_maze(int *rows, int *cols);
```

Reads the maze from `cin`. Allocates a 2D array for the maze and returns it. The `rows` and `cols` arguments should point to variables that can be filled in with the dimensions of the maze read in.

```
void print_maze(char **maze, int rows, int cols);
```

Prints the maze dimensions and maze contents to the `cout` in a two dimensional format. (Your completed program will call this after filling the shortest path with asterisks.)

queue.cpp

Your `queue.cpp` / `queue.h` files will define the `Location` struct as well as the following API for the `Queue` class:

```
         Queue(int max_size);              // constructor
        ~Queue();                          // destructor
    void Queue::add_to_back(Location loc); // add new locati
Location Queue::remove_from_front();       // get oldest loc
    bool Queue::is_empty();                // am I empty?
```

Most of the class definition will be provided for you. You will have to fill in the body of the `add_to_back` and `remove_from_front` functions. Below, we will explain the type of implementation that you should use, and give you code to help test it.

maze.cpp

Your `maze.cpp` should be broken modularly into two functions:

```
int maze_search(char **maze, int rows, int cols)
```

Given a maze, performs the BFS search for a valid path, filling in the path with `*` characters if found. Returns 1 if a valid path was found, 0 if no path was found, and -1 if the input is of the wrong format. Specifically, it should check that the input contains **exactly one start (`S`) cell and exactly one finish (`F`) cell**.

```
int main()
```

Starts the program, calls the input function, solves the maze, prints the output.

Keep in mind the following restrictions:

- You should exclusively be using dynamically-allocated arrays of exactly the right size this week. You will get a deduction if you just use a "large enough" statically allocated array of size 1000-by-1000.
- A valid path consists of steps north, south, east, and west but no diagonals. Thus we only need to explore neighbors of a cell in those 4 directions, not along diagonals.

BFS Pseudocode, Memory, Backtracking

Here is a review of the BFS description given earlier:

```
add start location to q
while q is not empty do
   let loc = extract oldest item from front of q
   for each neighbor of loc
      if neighbor is open and not yet explored
         add neighbor to back of q
         let predecessor of neighbor = loc
```

You need to avoid adding any location to the queue more than once. Otherwise, your search can cycle infinitely or exceed the maximum queue size. How can we do this?

Here is a *bad* way to solve this problem: before inserting a location, search the whole internal array of the `Queue` to see if it's already there. This could be made to work correctly, but it has *slow* performance, O(NROWS^2*NCOLS^2) worst-case time. **Don't do this!**

Instead, maintain a data structure that remembers, for each grid cell, if it's already been added to the queue or not. Therefore, this data structure should let you look up, for a given pair of row/col indices, whether that cell has already been visited, or not yet visited. What type of structure could you use for this? In your code, allocate and initialize this structure before you start the BFS algorithm. At what step of the BFS algorithm do you have to mark a cell as visited?

The final step is actually locating the optimal path and marking it with `*` characters. This requires a little more bookkeeping, like a "trail of breadcrumbs" that you can follow from the finish back to the start. This is the *predecessor* array referred to earlier. It should be a 2D array of `Location`s, so that for a given `[row][col]`, you can look up which position was its predecessor (which neighbor caused it to be added to the queue). The predecessor is utilized to find the actual path when your algorithm is complete: we trace it from the finish back to the start. I.e., *the predecessor of the finish cell will tell us how to go back one step, then that cell's predecessor will tell us how to go back another step, etc:*

```
previous_loc = predecessor[current_loc.row][current_loc.col]
```

- At what step of the algorithm will you fill a new entry in the predecessor array?
- Where should the backtracking start and finish?
- What kind of loop should you use for backtracking?

Prelab

The following is optional (not to be handed in) but strongly recommended (because it makes it much clearer what your program should be doing).

BEFORE beginning, consider the sample maze shown below and put yourself in place of the BFS. Our examples will explore each cell's neighbors in the order {North, West, South, East}. Show the coordinates of each cell placed in the Queue in the appropriate order from the Start node and stopping as soon as the 'Finish' node is entered into the queue. Then for each cell placed in the BFS queue, keep track of how the predecessor array would be updated. Show the predecessor array's FINAL value at the end of the BFS execution. We have started the example for you.

Maze:

```
   Col: 0123
 Row 0: ..#.
 Row 1: ..#S
 Row 2: F.#.
 Row 3: ....
```

Queue — items are `(row,col)`:

```
 tail: 4
 head: 3
 contents: [(1,3)|(0,3)|(2,3)|(3,3)|     |     |     | ... ]
```

Predecessor:

```
   Col:  0  |  1  |  2  |  3
 Row 0:     |     |     |(1,3)
 Row 1:     |     |     |
 Row 2:     |     |     |(1,3)
 Row 3:     |     |     |(2,3)
```

1. Complete this partially-started example until the queue is complete (or you can stop when hitting the finish).
2. Do the backtracking on this example. Which cell is the finish? If you backtrack from there, what path through the maze do you get?

Whether or not you do this optional example, **fill in** the `readme.txt` file prelab, which asks you about your design of the structure that you use to remember which cells have been visited or not.

Obtaining the Files

Create a directory anywhere you like, and inside of it, run

```
 wget ftp://bits.usc.edu/cs103/amaze.tar
 tar xvf amaze.tar
```

This contains the three .cpp files and two .h files for the project, the `readme.txt` template, the Makefile, the queue_test program (see below), and 4 sample mazes. #1 and #2 are normal mazes. #3 is an invalid maze. #4 is a valid maze with no path from the start to the finish.

Writeup and Submission

The `readme.txt` file asks you a couple of questions about the design of your algorithm.

As part of testing your code, you should create your own maze called `mymaze.txt`. Specifically, we want you to create a maze where there are multiple paths from the start to finish of different length. In this way, you can verify that your algorithm is actually computing a *shortest* path.

Submit the assignment using the link on the coursework page. Continue to follow the style guidelines from previous weeks.

Note that you do not have to submit the .h files, since those should not have been changed at all.

Q&A

Q: Do I have to account for empty mazes, or wrongly-shaped mazes?
A: No, you can assume the maze data forms a rectangle with dimensions matching the given integers, containing only the characters `.#SF`. You can assume the number of rows and columns is a positive integer.

Possible Progress Steps

The following is just a set of suggestions as to how you can develop the code while keeping the debugging work as simple as possible. You do not have to follow it.

Checkpoint 1: Input/output

Complete the `mazeio.cpp` code so that you can read/write mazes. In `main()` in `maze.cpp`, write a program that simply reads the input and prints it to the screen without trying to solve it.

Compile and run your program:

```
$ make maze
$ ./maze < maze1.in
5 5
.S.#.
##.#.
.....
.####
....F
```

Check that the printed output looks correct.

Make sure to deallocate the memory. Test this with `valgrind`.

Checkpoint 2: Queue

You will see two blank portions in `queue.cpp` where you need to fill in the blanks. These should be just a couple of lines long. One is in `add_to_back`, which should add the incoming `Location` to the end of the list. The other is `remove_from_front`, which should return the `Location` that has been waiting the longest to be removed. Each of these functions should update the appropriate counter. For help with the syntax, see the description of the member variables in `queue.h` and also look at how `is_empty` is defined in `queue.cpp`.

Use `make queue_test` to compile the test code given earlier in the assignment. Run it with `./queue_test` and see if the output is correct.

Checkpoint 3: Core BFS algorithm

Add the `#include "queue.h"` header to `maze.cpp`.

Now write the search code in `maze_search()`.

1. Find the start and finish cells and check that the maze is valid.
2. Setup and initialize your queue, predecessor and any other arrays/data structures necessary
3. Perform the Breadth First Search until the queue is empty.
4. Return the correct status code: 1 for success, 0 for no path exists, -1 for a badly-formatted maze.

Does your program successfully distinguish mazes where a path exists, from ones where a path does not? You should be able to run it on all of the test cases.

If need be, print out the location of each item you add or remove to the queue, for debugging purposes.

Final Product

1. Now add the code to walk the predecessor array backwards to the start location, filling in the cells with `*`
2. Make sure your code meets all the requirements given earlier in this handout.
3. Make sure all dynamically allocated memory is deallocated.
4. Compile and run your program on all available sample cases, as well as the one you created yourself. The more testing, the better.

Here is one sample run:

```
$ make maze
$ ./maze < maze1.in
5 5
.S*#.
##*#.
***..
*####
****F
```

Assignment created by Mark Redekopp and modified by David Pritchard.