CSE240 – Assignment 2
Dynamic Arrays

50 points

Topics:

- Created and used variables
- Used printf and scanf to work with input and output
- Use simple File I/O
- Used control structures to control the flow of logic
- Work with math and random numbers
- Dynamic Allocation of multi-dimension arrays
- Create some basic void and value returning functions

Description

The aim of this assignment is to make sure you understand dynamic array allocation by creating dynamic arrays and performing algorithms on them.

You have **two** different options. Pick ONE.

Use the following Guidelines:

- Give identifiers semantic meaning and make them easy to read (examples numStudents, grossPay, etc).
    - Keep identifiers to a reasonably short length.
    - Suggestions: Use upper case for constants. Use title case (first letter is upper case) for classes. Use lower case with uppercase word separators for all other identifiers (variables, methods, objects).
    - Strive for self-commenting code
- Use tabs or spaces to indent code within blocks (code surrounded by braces). This includes classes, methods, and code associated with ifs, switches and loops. Be consistent with the number of spaces or tabs that you use to indent.
- Use white space to make your program more readable.

Important Note:

All submitted assignments must begin with the descriptive comment block. To avoid losing trivial points, make sure this comment header is included in every assignment you submit and that it is updated accordingly from assignment to assignment.

```
/*
Name: <your name>
Date: <turn in date>
Description: <short description of code in file/project>
Usage: <how to use your program, including syntax for launching the program (command line
arguments)>
*/
```

Programming Assignment:

Instructions:

In this assignment you will create your code from scratch. You are to create a C/C++ file named <lastname>_<firstname>_hw2.c (or .cpp)

Stay within bounds of what we've covered in class. You are to use functions in this assignment.

All of the algorithms should be coded by you. Code the algorithms from scratch. Don't just find a library to solve the problems for you.

Library allowances:

If you are using C++ you may use: #include <stack> and std::stack or #include<queue> and std::queue for your algorithm needs.

Note: C does not have standardized data structures libraries. If you need those data-structures for your algorithms, you should code in C++ or build your own functionality via an array.

Specifications:

Option #1 – Wavefront Path Finder

Moving an artificial agent (robot or otherwise) generally takes three things:

1. A roadmap
2. A path
3. A steering algorithm

Different disciplines in AI have emerged to address each issue. Path planning is one name for the field which helps define item #1 – how do I translate an environment into a navigable space?

One such algorithm is the Wavefront or Grassfire Algorithm.

Set-Up

This algorithm starts with a grid representation of the environment (2D array). In this grid, obstacles are marked with a value of -1 to indicate they are non-navigable. Zeroes indicate unexplored or possibly unreachable positions.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	0	0	0	0
4	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Figure 1-Setup for WaveFront (this example uses 1 for obstacles)

Making the Wave

You take a starting point (which actually represents our goal position) and then "make a splash." The starting point is set to a value of 1.

Now you spread from that starting point storing an increasing value with each step away from the starting point.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	0	0	0	0
4	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

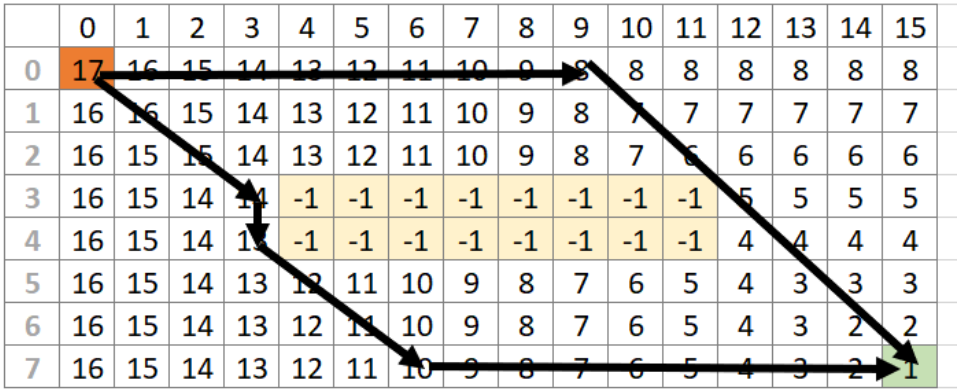
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	0	0	0	0
4	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	3	3	3
6	0	0	0	0	0	0	0	0	0	0	0	0	0	3	2	2
7	0	0	0	0	0	0	0	0	0	0	0	0	0	3	2	1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	7	7	7	7	7	7
2	0	0	0	0	0	0	0	0	0	0	7	6	6	6	6	6
3	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	5	5	5	5
4	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	4	4	4	4
5	0	0	0	0	0	0	0	0	0	7	6	5	4	3	3	3
6	0	0	0	0	0	0	0	0	0	7	6	5	4	3	2	2
7	0	0	0	0	0	0	0	0	0	7	6	5	4	3	2	1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	17	16	15	14	13	12	11	10	9	8	8	8	8	8	8	8
1	16	16	15	14	13	12	11	10	9	8	7	7	7	7	7	7
2	16	15	15	14	13	12	11	10	9	8	7	6	6	6	6	6
3	16	15	14	14	-1	-1	-1	-1	-1	-1	-1	-1	5	5	5	5
4	16	15	14	13	-1	-1	-1	-1	-1	-1	-1	-1	4	4	4	4
5	16	15	14	13	12	11	10	9	8	7	6	5	4	3	3	3
6	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	2
7	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Finding the Path

You continue the wave until every reachable position is filled. Now we have a way to find paths from the starting position to the goal. To find the path, pick a starting location then make ***greedy*** choices to find the goal point.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	17	16	15	14	13	12	11	10	9	8	8	8	8	8	8	8
1	16	16	15	14	13	12	11	10	9	8	7	7	7	7	7	7
2	16	15	15	14	13	12	11	10	9	8	7	6	6	6	6	6
3	16	15	14	14	-1	-1	-1	-1	-1	-1	-1	-1	5	5	5	5
4	16	15	14	13	-1	-1	-1	-1	-1	-1	-1	-1	4	4	4	4
5	16	15	14	13	12	11	10	9	8	7	6	5	4	3	3	3
6	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	2
7	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

The greedy choice is to always pick the smallest (non-obstacle) number adjacent to the current position.

Directions:

Set-Up

Welcome the user to your Wavefront Pathfinder.

You will ask the user for the size of the grid they want to generate.

```
Welcome to CSE240 WaveFront Pather

Please tell me about the grid you want to generate.
Enter the width (at least 10):
Enter the height (at least 10):
```

Based on the width and height you will **dynamically allocate** a 2D integer array. Initialize all the indexes to 0 at first.

Then ask the user how much of the environment should be impassible.

```
Enter the percent of impassible terrain (0 - 100):
```

Take that percentage as an integer. If the percentage is greater than 40 percent, warn the user that it might make for poor results and confirm the decision. If the user confirms, continue; otherwise, treat it as a bad input and re-prompt them.

```
Enter the percent of impassible terrain (0 - 100): 55

Having a value greater than 40% might create poor results, do you with to continue? (y/n):
```

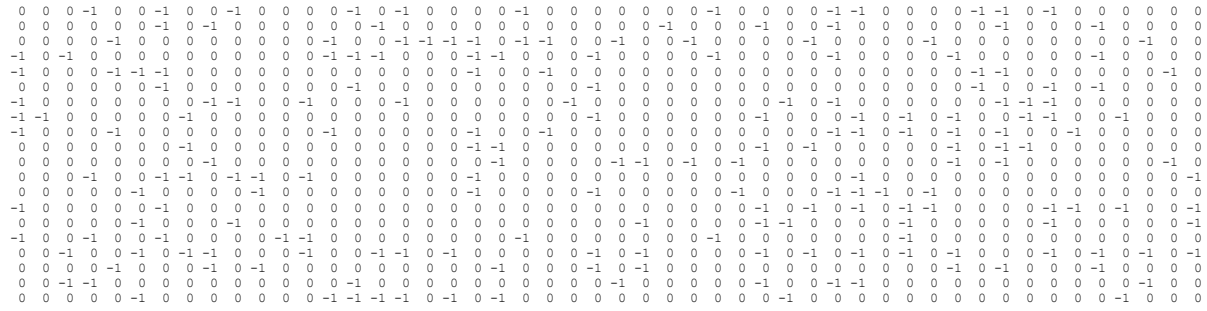
Generate Environment

Based on the percentage you took in from the user, randomly pick that many indexes in your 2D array to be set to -1.

Example 10x10 with 20% impassible:

```
  0   0   0   0 -1 -1   0 -1   0 -1
 -1   0   0   0   0 -1 -1   0   0   0
  0   0   0   0   0 -1   0   0   0   0
  0 -1   0   0   0   0   0   0   0   0
  0   0   0 -1   0   0   0   0   0 -1
  0   0 -1 -1   0   0   0   0   0   0
  0   0   0   0   0 -1   0   0   0 -1
  0 -1   0   0 -1 -1   0   0 -1   0
  0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0 -1   0   0   0
```

Example 50x20 with 20% impassible:



Create a function named void PrintEnvironment(int** array, int width, int height) that outputs the array to the screen using printf or <iomanip> to make it look nice.

Call this function to print the environment array after it is generated.

Get Goal Position

Ask the user for an x and y value to indicate the goal position. This goal also acts as the start of the wave.

```
Please enter the Goal Position X:
Please enter the Goal Position Y:
```

If the [x][y] position has a -1 in it, or if the input is out of bounds re-prompt the user

```
Please enter the Goal Position X: 5
Please enter the Goal Position Y: 0

Sorry, that position is inside an obstacle

Please enter the Goal Position X:
```

After getting that goal position, set that index in the environment array to 1 to set up for the Wave.

Run the Wave

For each index that is value 0 adjacent (up, down, left, right and diagonal) to change the value of indexes adjacent to one greater than the set index.

```
0 0 0          0 3 3
0 0 2   ---->  3 3 2
0 2 1          3 2 1
```

The wave algorithm is similar to a Flood-Fill algorithm. The difference is that you are filling the environment with increasing numeric values.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	17	16	15	14	13	12	11	10	9	8	8	8	8	8	8	8
1	16	16	15	14	13	12	11	10	9	8	7	7	7	7	7	7
2	16	15	15	14	13	12	11	10	9	8	7	6	6	6	6	6
3	16	15	14	14	-1	-1	-1	-1	-1	-1	-1	-1	5	5	5	5
4	16	15	14	13	-1	-1	-1	-1	-1	-1	-1	-1	4	4	4	4
5	16	15	14	13	12	11	10	9	8	7	6	5	4	3	3	3
6	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	2
7	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

There are multiple approaches to a flood-fill algorithm. Some are better than others.

The very first time I ever tried my hand at this problem, my algorithm was O(n^3) … really bad. Essentially I processed the entire NxM array looking for cells with value 0 with neighbors, then storing that neighbor value+1 in the cell. After I ran through the whole array, repeat!

Some major approaches:

- N-Way (generally recursive) – O(n^2)
- Linescan flood fill – O(n^2)
- Breadth-First Search – O(|E| + |V|)
    - {|E| is the number of edges, |V| is the number of points}

Difference between the N-Way and the BFS? The BFS basically spirals out from the point that the user selects and N-Way "blobs out" in that direction.

After running your flood/wave output the environment array to the screen.

Finding the Path and Finishing the Job

Dynamically create a character array of the same width and height as your environment array.

This character array should be parallel to your environment array. Wherever there is a -1 in your environment array, put a '#' in the character array.

Example:

```
  0  0  0  0  0  0  0  0 -1  0
  0 -1 -1  0  0  0  0  0  0  0
  0  0  0  0  0  0 -1  0  0 -1
  0  0  0  0  0 -1  0 -1  0  0
  0  0 -1  0  0 -1  0  0  0  0
 -1  0  0  0  0  0  0  0  0  0
  0  0  0 -1 -1  0 -1  0  0  0
 -1  0 -1 -1  0  0  0  0 -1  0
  0  0  0  0  0  0  0  0  0  0
  0  0 -1  0 -1  0  0  0  0 -1
```

Result array:

```
        #
 ##
      #  #
     # #
   #  #
#
   ## #
# ##    #

  # #    #
```

Prompt the user for a starting point for the path finder.

```
Please enter the Start Position X:
Please enter the Start Position Y:
```

If the [x][y] position has a -1 in it, or if the input is out of bounds re-prompt the user

```
Please enter the Start Position X: 2
Please enter the Start Position Y: 1

Sorry, that position is inside an obstacle

Please enter the Start Position X:
```

Mark the start position chosen in the character array with an '@' and mark the goal position with a '$'

From the starting point, perform the greedy pathing algorithm.

Look in the immediate surrounding 8 cells and choose the next cell with the smallest value.

Place an * in the character array to represent the path chosen.

You do not have to find ALL paths, just one.

Output the character array including the found path to the screen.

Result array:

```
@***    #
 ## *
     *#  #
     #*#
   #  #*
#       *
   ## # *
# ##    #*
         *
  # #    #$
```

Into a file named wave.txt output first the integer environment array, then two new lines, then the finished character array.

End the program after all output and file output have been completed.

Interface:

Please ask for input in this order. I'm trying to standardize interfaces to help streamline grading. You can change the display of the interface and customize it, but the information should come in this order.

You should welcome the user then prompt them for data.

```
Welcome to <your name here>'s CSE240 WaveFront Pather


Enter the width (at least 10):
Enter the height (at least 10):
Enter the percent of impassible terrain (0 - 100):
Please enter the Goal Position X:
Please enter the Goal Position Y:
Please enter the Start Position X:
Please enter the Start Position Y:
```

Once again, this is just the script for the inputs. It is not exhaustive. I'm leaving out defensive checks and re-prompts. You can also customize the wording.

Extra Credit Opportunity +5:

Give the user an option to provide a file for input. This should be indicated by the user running the program with the command-line argument -f. When -f is present, the user should skip the initial environment set up and instead be prompted for the environment file to read from.

This file should be formatted this way:

<number of rows>

<number of columns>

<grid of 0's and 1's to indicate initial environment (spaces in between)>

1's should be considered obstacles and converted into -1 for the environment array as indicated in the rest of the spec.

User interface should continue from Goal Position prompts

Option #2 – Procedural Island Generator

This option has you creating an island/continent out of a method known as a "Particle Roll" Algorithm.

The idea is simple:

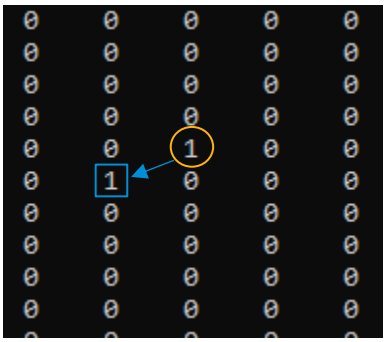
Set-Up:

1. Get a width and height from the user
2. Create a dynamically allocated 2D Array based on that width and height
3. Get an (x, y) position and width and height to define a "drop window".
    a. windowX & windowY >= 0
    b. windowWidth & windowHeight >= 1
        - windowXMax = windowX + windowWidth
        - windowYMax = windowY + windowHeight
4. Get the number of particles to drop from the user.
5. Get the max-life from the user
6. Get a value between 40 and 200 to use as the water-line

Drop-Roll-Generation:

1. "drop" a particle at a random x,y location within the window
2. increment that index by 1
3. confirm there is a valid move the particle can make
    a. the adjacent index value <= the current index value
    b. the adjacent index is >= 0 and < width/height
4. pick a random valid move and make it
    a. change the particle's x,y index to that new spot
5. reduce the particle's life
    a. max-life is the number of moves the particle can make
6. repeat 2-5 until the particle runs out of life or there are no valid moves

Visualization:



The Yellow Circle represents the original drop point

The Blue Arrow the move that was chosen

The Blue Box the new position (which is also incremented). The new position used to be 0 and is now 1 because the particle 'rolled into it.'

When a particle "rolls" one space, it loses one life. If your particles have a max-life of 50, they can move 50 times before stopping.

Visualization Part 2:

Here is an example of a single particle with a max-life of 50.

You can see the path it generally traveled. Some indexes became 2 or 3 because the particle was able to roll into those positions multiple times.

```
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   1   1   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   1   1   1   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   1   0   0   2   1   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   1   2   1   2   2   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   2   1   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   1   2   3   1   1   2   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   1   0   2   1   3   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   1   1   1   1   3   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   1   0   1   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

Visualization Part 3:

500 particles, max-life of 50.

```
  0   1   0   0   0   1   0   0   0   0   0   1   2   0   0   0   0   0   0   0   0   0   0   0   0
  0   2   1   0   0   1   1   0   0   0   0   1   1   3   0   0   1   0   0   0   0   0   0   0   0
  1   1   1   0   0   2   1   0   0   2   2   4   4   0   2   2   3   1   0   0   0   0   0   0   0
  0   2   3   0   0   2   3   4   5   4  11   3   1   3   4   3   2   1   1   0   0   0   0   0   0
  0   2   2   4   3   3   5   7   4  13  10   7   6   3   5   4   2   3   1   0   0   0   0   1   4
  0   1   1   2   3   6   8  10  10  12   8  13   6   6   5   4   2   1   1   1   0   1   0   5   2
  0   1   0   0   2   5   9   9   8  13  14  15  13  11   5   6   7   1   3   3   1   2   3   1   0
  0   0   4   3   6   7  10  11  13  14  20  19  14  11  10  12  11  10   7   2   1   1   2   2   0
  0   4   4   2   6   7  10  14  20  25  22  25  24  22  17  20  14  10   8   5   3   1   1   2   0
  0   1   5   6   5   9   6  10  27  28  23  23  26  25  25  19  16  12   7   6   4   1   1   4   0
  3   4   6   5   6  10  11  17  28  29  29  25  27  26  23  25  18  10   7   4   6   2   0   1   0
  3   3   2   5   5   7  17  18  28  28  26  23  23  28  30  28  27   7  10   7   5   4   2   1   0
  0   4   2   4   5  10  15  18  24  21  22  16  25  26  29  29  21  16   9   8   6   4   3   0   2
  2   0   3   2   8   9  12  15  21  25  21  20  28  28  27  23  22  15  12   7   6   5   1   1   1
  1   3   3   3   1   6  10  15  22  25  22  23  27  27  24  25  21  17  13   8   5   2   1   2   2
  2   2   1   2   1   6   2  10  23  21  22  27  26  27  28  22  20  12  11   9   6   1   1   4   2
  2   2   0   0   3   5   6   8  11  18  21  24  23  23  21  20  15  11  10   7   2   2   3   3   1
  1   1   1   1   2   1   1   7   8  10  14  16  18  19  16  15  13  12  10   5   2   4   4   5   4
  0   0   0   2   2   0   1   1   2   8  10  11   8   8  14  11  10   9   7   4   5   6   1   4   2
  0   0   0   3   3   2   0   0   6   8   3   7   7   8   6   9  11   5   4   5   2   5   4   1   1
  0   1   0   3   4   3   2   0   0   1   2   4   5   5   6   4   6   5   3   2   4   4   5   1   0
  0   0   2   4   5   0   0   0   0   0   1   4   3   3   2   6   5   4   3   3   0   3   2   2   0
  0   0   1   3   1   0   0   0   1   2   1   2   0   0   4   5   2   0   0   2   2   0   1   0   0
  0   4   2   1   1   0   0   1   2   1   2   1   1   2   2   2   0   2   0   1   0   0   0   0   0
  1   4   3   1   1   1   1   1   0   2   1   0   2   3   1   0   1   0   0   0   1   0   0   0   0
```

Polishing the Island

After you have generated the values for the landmass, do a quick analysis to find the highest value in the 2D array.

Normalize the values in the 2D array by dividing every value in the 2D array by that maximum (don't forget to use floating point division) then multiplying by 255. This will make every index a value between 0 and 255.

Create a 2D character array of the same size as your value array.

Process through the values in your array and set a character in your 2D character array based on the value's relationship to the water-line (from the set-up).

Output your polished island to the console AND to a file (island.txt is fine).

Calculate land-zone as 255 – water-line.

- value < 50% of water-line
  - '#' – deep water
- value > 50% of water-line && <= water-line
  - '~' – shallow water

Everything else is > water-line &&

- < (water-line + 15% of land-zone)
  - '.' - coast/beach
- >= (water-line + 15% of land-zone) && < (water-line + 40% of land-zone)
  - '-' – plains/grass
- >= (water-line + 40% of land-zone) && < (water-line + 80% of land-zone)
  - '*' – forests
- else:
  - '^' – mountains

Island Normalized:

```
  0   8   0   0   0   8   0   0   0   0   0   8  17   0   0   0   0   0   0   0   0   0   0   0   0
  0  17   8   0   0   8   8   0   0   0   0   8   8  25   0   0   8   0   0   0   0   0   0   0   0
  8   8   8   0   0  17   8   0   0  17  17  34  34   0  17  17  25   8   0   0   0   0   0   0   0
  0  17  25   0   0  17  25  34  42  34  93  25   8  25  34  25  17   8   8   0   0   0   0   0   0
  0  17  17  34  25  25  42  59  34 110  85  59  51  25  42  34  17  25   8   0   0   0   0   8  34
  0   8   8  17  25  51  68  85  85 102  68 110  51  51  42  34  17   8   8   8   0   8   0  42  17
  0   8   0   0  17  42  76  76  68 110 119 127 110  93  42  51  59   8  25  25   8  17  25   8   0
  0   0  34  25  51  59  85  93 110 119 170 161 119  93  85 102  93  85  59  17   8   8  17  17   0
  0  34  34  17  51  59  85 119 170 212 187 212 204 187 144 170 119  85  68  42  25   8   8  17   0
  0   8  42  51  42  76  51  85 229 238 195 195 221 212 212 161 136 102  59  51  34   8   8  34   0
 25  34  51  42  51  85  93 144 238 246 246 212 229 221 195 212 153  85  59  34  51  17   0   8   0
 25  25  17  42  42  59 144 153 238 238 221 195 195 238 255 238 229  59  85  59  42  34  17   8   0
  0  34  17  34  42  85 127 153 204 178 187 136 212 221 246 246 178 136  76  68  51  34  25   0  17
 17   0  25  17  68  76 102 127 178 212 178 170 238 238 229 195 187 127 102  59  51  42   8   8   8
  8  25  25  25   8  51  85 127 187 212 187 195 229 229 204 212 178 144 110  68  42  17   8  17  17
 17  17   8  17   8  51  17  85 195 178 187 229 221 229 238 187 170 102  93  76  51   8   8  34  17
 17  17   0   0  25  42  51  68  93 153 178 204 195 195 178 170 127  93  85  59  17  17  25  25   8
  8   8   8   8  17   8   8  59  68  85 119 136 153 161 136 127 110 102  85  42  17  34  34  42  34
  0   0   0  17  17   0   8   8  17  68  85  93  68  68 119  93  85  76  59  34  42  51   8  34  17
  0   0   0  25  25  17   0   0  51  68  25  59  59  68  51  76  93  42  34  42  17  42  34   8   8
  0   8   0  25  34  25  17   0   0   8  17  34  42  42  51  34  51  42  25  17  34  34  42   8   0
  0   0  17  34  42   0   0   0   0   0   8  34  25  25  17  51  42  34  25  25   0  25  17  17   0
  0   0   8  25   8   0   0   0   8  17   8  17   0   0  34  42  17   0   0  17  17   0   8   0   0
  0  34  17   8   8   0   0   8  17   8  17   8   8  17  17  17   0  17   0   8   0   0   0   0   0
  8  34  25   8   8   8   8   8   0  17   8   0  17  25   8   0   8   0   0   0   8   0   0   0   0
```

Polished

Figure 1 Waterline = 70

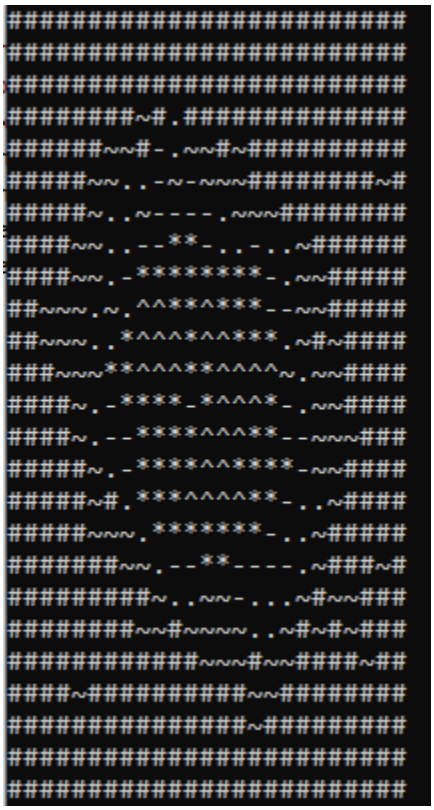


Figure 2 Waterline = 10

Interface:

Please ask for input in this order. I am trying to standardize interfaces to help streamline grading. You can change the display of the interface and customize it to be more user friendly, but the information should come in this order.

You should welcome the user then prompt them for data.

```
Welcome to <your name here>’s CSE240 island generator!

Enter grid width:
Enter grid height:
Enter drop-window x-coordinate (0 - <width>):
Enter drop-window y-coordinate (0 - <height>):
Enter drop-window width (>= 1):
Enter drop-window height (>= 1):
Enter number of particles to drop:
Enter max life of particles:
Enter value for waterline (40-200):
```

Once again, this is just the script for the inputs. It is not exhaustive. I’m leaving out defensive checks and re-prompts. You can also customize the wording.

Recommended Functions

Quality modularization of your code is part of your Code-Quality score in the rubric.

These function recommendations are for your benefit to make organizing this project easier …

makeParticleMap

```
int** makeParticleMap(int width, int height,
                      int windowX, int windowY, int winWidth, int winHeight,
                      int numParticles, int maxLife)
```

This function builds the array for the map data and performs particle roll algorithm to populate the array with map data.

You can modify this function to take the array as a parameter instead if you prefer to create the array externally to the particle roll algorithm. In which case, this function would become a void type.

NOTE: windowX, windowY, winWidth, winHeight are the parameters for the drop-window.

moveExists

```
bool moveExists(int** map, int width, int height, int x, int y)
```

This function looks a the 8 spots around x,y and determines if a valid move is possible. Return true if one is found. Note: you can return true as soon as the first one is found.

findMax

```
int findMax(int** map, int width, int height)
```

This function finds the maximum value in the map and returns it.

normalizeMap

```
void normalizeMap(int** map, int width, int height, int maxVal)
```

Performs the normalization operation on the map data. You could have this return a new array instead if you want to hold onto the original map data for some reason.

Extra Credit Opportunities:

Special Interface +3

Take a command line argument -S which gives the user access to an expanded user interface. If the -S doesn't exist, use the default interface as described above.

Create a special interface to let the user repeatedly drop particles on a new location instead of a single drop.

The repeated drops should allow the user to define a new drop-window, number of particles and max life.

Polish the map after they are done making drops.

NOTE: to get credit for this, we need to know it is there. Please include something to indicate the -S interface is available in your Welcome Message when the program loads.

Color +2

Color your output using a text coloring library. Find a console-color library that works across all Operating Systems and integrate it.

NOTE – this should be stand alone and not require any installation on the grader's part, just the library files in the proper location.

Any required compilation should be accounted for in a Makefile.

MAKE SURE you put any instruction for building/running your code in your submission notes.

I have used: https://github.com/ikalnytskyi/termcolor in the past to good effect.

Bugs and Debugging:

You should work within reasonable I/O standards. If the user gives you a number outside the parameters, you should correct the user and loop the input prompt again.

I will not hold you responsible for bizarre input like text into an integer at this point.

Notes and tips:

- Remember to write Functions. Since there are so many options and algorithms here it is impossible for me to write Required functions.
  - Modularize your code wherever and whenever possible!!
  - Remember code readability is a thing!
  - Don't throw away coding standards for the sake of the algorithm.
    - If you are copying the algorithm in someone else's coding style, that's a big **red flag** that you need to redo the work until you understand the algorithm and can code it yourself in your style
- Code the algorithms yourself! Research each as necessary and accomplish the goals
  - The flood fill algorithm is "old" and widely available on the internet, we will be watching for copied code
  - Study the algorithms and code your own
  - Study the scenario and try to puzzle it out yourself (even better)
- Generating a random number is easy in C, but you will want to include the time library to seed the random number generator. You might want to use this for testing your code.
  - `#include <time.h>`
  - `srand(time(NULL)); //only needed once for your whole program usually in main`
  - `value = (rand() % range_value) + minimum_value;`
- Remember, I'm allowing you to use some of the standard data-structures libraries for this assignment.
  - Make sure you look them up and understand how to use them properly

Grading of Programming Assignment

The Grader will grade your program following these steps:

(1) Compile the code. If it does not compile a U or F will be given in the Specifications section. This will probably also affect the Efficiency/Stability section.

(2) The Grader will read your program and give points based on the points allocated to each component, the readability of your code (organization of the code and comments), logic, inclusion of the required functions, and correctness of the implementations of each function.

Rubric:

	Levels of Achievement						
Criteria	A	B	C	D	E	U	F
Specifications Weight 50.00%	100 % The program works and meets all of the specifications.	85 % The program works and produces the correct results and displays them correctly. It also meets most of the other specifications.	75 % The program produces mostly correct results but does not display them correctly and/or missing some specifications	65 % The program produces partially correct results, display problems and/or missing specifications	35 % Program compiles and runs and attempts specifications, but several problems exist	20 % Code does not compile and run. Produces excessive incorrect results	0 % Code does not compile. Barely an attempt was made at specifications.
Code Quality Weight 20.00%	100 % Code is written clearly	85 % Code readability is less	75 % The code is readable only by someone who knows what it is supposed to be doing.	65 % Code is using single letter variables, poorly organized	35 % The code is poorly organized and very difficult to read.	20 % Code uses excessive single letter identifiers. Excessively poorly organized.	0 % Code is incomprehensible
Documentation Weight 15.00%	100 % Code is very well commented	85 % Commenting is simple but solid	75 % Commenting is severely lacking	65 % Bare minimum commenting	35 % Comments are poor	20 % Only the header comment exists identifying the student.	0 % Non existent
Efficiency Weight 15.00%	100 % The code is extremely efficient without sacrificing readability and understanding.	85 % The code is fairly efficient without sacrificing readability and understanding.	75 % The code is brute force but concise.	65 % The code is brute force and unnecessarily long.	35 % The code is huge and appears to be patched together.	20 % The code has created very poor runtimes for much simpler faster algorithms.	0 % Code is incomprehensible

What to Submit?

You are required to submit your solutions in a compressed format (.zip). Zip all files into a single zip file. Make sure your compressed file is labeled correctly - <lastname>_<firstname>_hw2.zip

The compressed file MUST contain the following:

- <lastname>_<firstname>_hw2.c / .cpp

If you did extra credit that involved File I/O, please include a sample file. No other files should be in the compressed folder.

If multiple submissions are made, the most recent submission will be graded, even if the assignment is submitted late.

Where to Submit?

All submissions must be electronically submitted to the respected homework link in the course web page where you downloaded the assignment.

Academic Integrity and Honor Code.

You are encouraged to cooperate in study group on learning the course materials. However, you may not cooperate on preparing the individual assignments. Anything that you turn in must be your own work: You must write up your own solution with your own understanding. If you use an idea that is found in a book or from other sources, or that was developed by someone else or jointly with some group, make sure you acknowledge the source and/or the names of the persons in the write-up for each problem. When you help your peers, you should never show your work to them. All assignment questions must be asked in the course discussion board. Asking assignment questions or making your assignment available in the public websites before the assignment due will be considered cheating.

*The instructor and the TA will **CAREFULLY** check any possible proliferation or plagiarism. We will use the document/program comparison tools like MOSS (Measure Of Software Similarity: http://moss.stanford.edu/) to check any assignment that you submitted for grading. The Ira A. Fulton Schools of Engineering expect all students to adhere to ASU's policy on Academic Dishonesty. These policies can be found in the Code of Student Conduct:*

http://www.asu.edu/studentaffairs/studentlife/judicial/academic_integrity.htm

ALL cases of cheating or plagiarism will be handed to the Dean's office. Penalties include a failing grade in the class, a note on your official transcript that shows you were punished for cheating, suspension, expulsion and revocation of already awarded degrees.