CS1010E Practical Assessment #3: Sweep (Question)

Tags & Categories

Tags:

Categories:

Related Tutorials

Task Content

Sweep!

Topic Coverage

- Assignment and expressions
- Control statements
- Functions and procedures
- Arrays

Problem Description

In the game of Minesweeper, the objective for the player is to clear the minefield without blowing up a mine. The player is presented with a `m` rows-by-`n` columns rectangular grid of cells under which some contain mines, and some do not. Here, we illustrate with a `16`-by-`30` minefield with `*` used to denote the presence of a mine. We use `.` to denote all other cells that are safe (i.e. do not contain mines).

```
..*...*.....*......**...**....
*.**.**....*.............*....
*.....*.*.*................*.*
.........***..............**.*
..........*..........*.....*..
.......*........*.....*.......
**...........*...*....*....*..
**.*...........*..*....**.....
**....*...*...........*.......
....*.*...*.......*........*.*
.................*.*...**..*..
.......*.*.**.........**......
..**..............*.*.*.*....*
..*.**...*.*.*.*..*........*.*
*..***.........*.*.........**.
........**....*...*.......*.**
```

Clearing a cell that is **safe** results in one of two situations:

- If there are mines surrounding the cell, then an integer value (`1` to `8`) representing the number of mines surrounding it is shown. For example, clearing the minefield at location `(14,6)` (i.e. row 14 and column `6`) would result in the following:

```
..*...*.....*......**...**....
*.**.**....*.............*....
*.....*.*.*................*.*
.........***..............**.*
..........*..........*.....*..
.......*........*.....*.......
**...........*...*....*....*..
```

```
**.*...........*..*....**.....
**....*...*...........*.......
....*.*...*.......*........*.*
.................*.*...**..*..
.......*.*.**.........**......
..**..............*.*.*.*....*
..*.**...*.*.*.*..*........*.*
*..***2........*.*.........**.
........**....*...*.......*.**
```

- If there are no mines surrounding the cell, then all eight neighbouring cells are also cleared. For example, clearing the minefield at location `(11,5)` would result in the following:

```
..*...*.....*......**...**....
*.**.**....*.............*....
*.....*.*.*................*.*
.........***..............**.*
..........*..........*.....*..
.......*........*.....*.......
**...........*...*....*....*..
**.*...........*..*....**.....
**....*...*...........*.......
....*.*...*.......*........*.*
....122..........*.*...**..*..
....101*.*.**.........**......
..**322...........*.*.*.*....*
..*.**...*.*.*.*..*........*.*
*..***.........*.*.........**.
........**....*...*.......*.**
```

In particular, if a neighbouring cell is itself not surrounded by mines, then it's neighbours are also cleared. As such, this may result in a progressively larger area being cleared. For example, clearing the minefield at location `(4,1)` would result in the following:

```
..*...*.....*......**...**....
*.**.**....*.............*....
*32223*.*.*................*.*
1100011..***..............**.*
0000001...*..........*.....*..
2210001*........*.....*.......
**31101......*...*....*....*..
**.*111........*..*....**.....
**....*...*...........*.......
....*.*...*.......*........*.*
.................*.*...**..*..
.......*.*.**.........**......
..**..............*.*.*.*....*
..*.**...*.*.*.*..*........*.*
*..***.........*.*.........**.
........**....*...*.......*.**
```

By repeatedly choosing safe cells to clear, larger parts of the minefield gets exposed. Below shows the state of the minefield upon clearing cells at locations `(14,6)`, `(11,5)`, `(4,1)`, `(0,14)`, and `(10,15)`.

```
..*...*.....*100001**101**....
*.**.**....*2100001221013*....
*32223*.*.*.20000000000012.*.*
1100011..***10000000111001**.*
0000001...*.100111001*21013*..
2210001*....1111*21013*2002...
**31101......*...*2102*4211*..
**.*111....1112*..*102.**.....
**....*...*200111..201*.......
....*.*...*200001.*211.....*.*
....122....321001*.*...**..*..
....101*.*.**1001.....**......
..**322......2211.*.*.*.*....*
..*.**...*.*.*.*..*........*.*
*..***2........*.*.........**.
........**....*...*.......*.**
```

Task

Write a program that reads an `m`-by-`n` minefield comprising of values `9` (representing a mine) and `-1` (safe). The program then repeatedly reads cell locations `(r, c)` (i.e. row `r` and column `c`) in which to clear and outputs the minefield after clearing the cells. The program terminates in one of three ways:

- Clearing the location of a cell with a mine;
- Quitting by clearing an invalid location; or
- When all safe cells are cleared.

Finally, the program outputs the minefield showing the mines, as well as the cells that were cleared.

Take note of the following:

- The top left location of the minefield is given by `(0, 0)`.
- The minefield will be no larger than `16` rows and `30` columns.
- To simplify program testing, sample runs provided comprise only small minefields. If time permits, you should test with bigger and more intricate minefields.
- Only one sample test case is provided to test for format correctness.
- Suggested function declarations are provided; you may choose to use, modify or ignore them.
- **Do not** use functions from the C Math library.

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.**

Level 1

Name your program `sweep1.c`

Write a program that reads two integers `m` and `n` representing the dimensions of the minefield.

The program then outputs the two values read.

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

```
$ ./a.out
3 4
3 4
```

Click here to submit to CodeCrunch.

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

```
./a.out < sweep.in | diff - sweep1.out
```

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

```
cp sweep1.c sweep2.c
```

Level 2

Name your program `sweep2.c`

Write a program that reads two integers `m` and `n` representing the dimensions of the minefield. The program then reads th
values of `9` (representing a mine) or `-1` (safe).

Finally, the program outputs the minefield. Use `*` to represent a mine, or `.` otherwise.

You may define the following functions:

`void readBoard(int board[][COL], int m, int n);` or
`void readBoard(int board[][COL], int *m, int *n);`
Read the minefield of dimensions `m` rows and `n` columns.

`void printBoard(int board[][COL], int m, int n);`
Output the minefield of dimensions `m` rows and `n` columns.

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

```
$ ./a.out
3 4
```

```
-1  9 -1  9
-1 -1 -1  9
-1 -1 -1 -1
.*.*
...*
....
```

Click here to submit to CodeCrunch.

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

```
./a.out < sweep.in | diff - sweep2.out
```

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

```
cp sweep2.c sweep3.c
```

Level 3

Name your program `sweep3.c`

Write a program that reads two integers `m` and `n` representing the dimensions of the minefield, followed by the minefield co
(representing a mine) or `-1` (safe).

The program then reads the location (row and column values) of **exactly one valid inner cell (i.e. not lying on the corn minefield)** to be cleared. If this cell does not contain a mine, then output the number of mines surrounding this cell. Other outputs the minefield. Use `*` to represent a mine or `.` to represent uncleared cells.

You may define the following functions:

`void readBoard(int board[][COL], int m, int n);` or
`void readBoard(int board[][COL], int *m, int *n);`
Read the minefield of dimensions `m` rows and `n` columns.

`void printBoard(int board[][COL], int m, int n);`
Output the minefield of dimensions `m` rows and `n` columns.

`int countNeighbour(int board[][COL], int r, int c);`
Count the number of mines in the neighbouring cells with respect to location `(r,c)`.

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

```
$ ./a.out
3 4
-1  9 -1  9
-1 -1 -1  9
-1 -1 -1 -1
1 2
.*.*
..3*
....
```

```
$ ./a.out
3 4
-1 -1 -1  9
-1 -1 -1  9
-1 -1 -1 -1
1 1
...*
.0.*
....
```

```
$ ./a.out
3 4
-1 -1 -1  9
-1 -1  9  9
-1 -1 -1 -1
1 2
...*
..**
....
```

Click here to submit to CodeCrunch.

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

```
./a.out < sweep.in | diff - sweep3.out
```

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

```
cp sweep3.c sweep4.c
```

Level 4

Name your program sweep4.c

Write a program that reads two integers `m` and `n` representing the dimensions of the minefield, followed by the minefield c
(representing a mine) or `-1` (safe).

The program then reads the locations (row and column values) of **any one valid cell** to be cleared. If this cell does not c
output the number of mines surrounding this cell. Otherwise, the program outputs the minefield. Use `*` to represent a min
uncleared cells.

You may define the following functions:

`void readBoard(int board[][COL], int m, int n);` or
`void readBoard(int board[][COL], int *m, int *n);`
  Read the minefield of dimensions `m` rows and `n` columns.

`void printBoard(int board[][COL], int m, int n);`
  Output the minefield of dimensions `m` rows and `n` columns.

`int countNeighbour(int board[][COL], int r, int c, int m, int n);`
  Count the number of mines in the neighbouring cells with respect to location `(r,c)`, while remaining within the `m`-b

`bool isValid(int r, int c, int m, int n);`
  Returns `true` if location `(r,c)` is within the `m`-by-`n` minefield, or `false` otherwise.

Tip: Make sure that each neighbour location is valid first, before attempting to access the minefield. What seems to work
not work on CodeCrunch.

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

```
$ ./a.out
3 4
-1  9 -1  9
-1 -1 -1  9
-1 -1 -1 -1
1 2
.*.*
..3*
....
```

```
$ ./a.out
3 4
-1  9 -1  9
-1 -1 -1  9
-1 -1 -1 -1
0 2
.*3*
...*
....
```

```
$ ./a.out
3 4
-1  9 -1  9
-1 -1 -1  9
-1 -1 -1 -1
0 0
1*.*
...*
....
```

```
$ ./a.out
3 4
```

```
-1  9 -1  9
-1 -1 -1  9
-1 -1 -1 -1
2 0
.*.*
...*
0...
```

```
$ ./a.out
3 4
-1  9 -1  9
-1 -1 -1  9
-1 -1 -1 -1
0 1
.*.*
...*
....
```

Click here to submit to CodeCrunch.

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

```
./a.out < sweep.in | diff - sweep4.out
```

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

```
cp sweep4.c sweep5.c
```

Level 5

Name your program sweep5.c

Write a program that reads two integers m and n representing the dimensions of the minefield, followed by the minefield c
(representing a mine) or -1 (safe).

The program then **repeatedly** reads cell locations (row and column values). If the cell is not a mine, then determine the n
surrounding this cell. The program terminates when one of the following conditions is met:

- An invalid cell location is read;
- The cell location contains a mine;
- All safe cells are cleared

Finally, the program outputs the minefield.

You may define the following functions:

`void readBoard(int board[][COL], int m, int n);` or
`void readBoard(int board[][COL], int *m, int *n);`
Read the minefield of dimensions m rows and n columns.

`void printBoard(int board[][COL], int m, int n);`
Output the minefield of dimensions m rows and n columns.

`int countNeighbour(int board[][COL], int r, int c, int m, int n);`
Count the number of mines in the neighbouring cells with respect to location (r,c), while remaining within the m-b

`bool isValid(int r, int c, int m, int n);`
Returns true if location (r,c) is within the m-by-n minefield, or false otherwise.

`bool allCleared(int board[][COL], int m, int n);`
Returns true if all safe locations of the m-by-n minefield have been cleared, or false otherwise.

Tip: Process the minefield each time a cell location is read; there is no need to store cell locations first.

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

```
$ ./a.out
3 4
-1 -1 -1  9
-1 -1 -1  9
-1 -1 -1 -1
1 2
```

```
0 1
4 4
.0.*
..2*
....
```

```
$ ./a.out
3 4
-1 -1 -1  9
-1 -1 -1  9
-1 -1 -1 -1
0 3
...*
...*
....
```

```
$ ./a.out
3 4
-1 -1 -1  9
-1 -1 -1  9
-1 -1 -1 -1
-1 -1
...*
...*
....
```

```
$ ./a.out
3 4
9  9  9 -1
9 -1  9  9
9  9  9  9
0 3
1 1
***3
*8**
****
```

Click here to submit to CodeCrunch.

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

```
./a.out < sweep.in | diff - sweep5.out
```

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

```
cp sweep5.c sweep6.c
```

Level 6

Name your program `sweep6.c`

Write a program that reads two integers `m` and `n` representing the dimensions of the minefield, followed by the minefield c
(representing a mine) or `-1` (safe).

The program then repeatedly reads the locations (row and column values) of any cell. If the cell is not a mine, then detern
mines surrounding this cell. **In the case where there is no surrounding mine, clear all its immediate neighbouring c**
terminates when one of the following conditions is met:

- An invalid cell location is read;
- The cell location contains a mine;
- All safe cells are cleared

Finally, the program outputs the minefield.

You may define the following functions:

`void readBoard(int board[][COL], int m, int n);` or
`void readBoard(int board[][COL], int *m, int *n);`
    Read the minefield of dimensions `m` rows and `n` columns.

`void printBoard(int board[][COL], int m, int n);`
    Output the minefield of dimensions `m` rows and `n` columns.

`int countNeighbour(int board[][COL], int r, int c, int m, int n);`
Count the number of mines in the neighbouring cells with respect to location `(r,c)`, while remaining within the `m`-b

`bool isValid(int r, int c, int m, int n);`
Returns `true` if location `(r,c)` is within the `m`-by-`n` minefield, or `false` otherwise.

`bool allCleared(int board[][COL], int m, int n);`
Returns `true` if all safe locations of the `m`-by-`n` minefield have been cleared, or `false` otherwise.

`void clearNeighbours(int board[][COL], int r, int c, int m, int n);`
Clears all immediate neighbour cells with respect to cell location `(r,c)`, while ensuring that the neighbours are wi

Tip: Make sure that each neighbour location is valid first, before attempting to access the minefield. What seems to work c not work on CodeCrunch.

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

```
$ ./a.out
3 4
-1 -1 -1  9
-1 -1 -1  9
-1 -1 -1 -1
1 2
1 3
...*
..2*
....
```

```
$ ./a.out
3 4
-1 -1 -1  9
-1 -1 -1  9
-1 -1 -1 -1
1 1
-1 -1
002*
002*
001.
```

```
$ ./a.out
3 4
-1 -1 -1  9
-1 -1 -1  9
-1 -1 -1 -1
1 1
2 3
002*
002*
0011
```

```
$ ./a.out
3 4
-1 -1 -1  9
-1 -1 -1  9
-1 -1 -1 -1
0 1
4 4
002*
002*
....
```

```
$ ./a.out
3 4
-1 -1 -1  9
-1 -1 -1  9
-1 -1 -1 -1
0 3
...*
...*
....
```

```
$ ./a.out
3 4
-1 -1 -1  9
-1 -1 -1  9
-1 -1 -1 -1
-1 -1
...*
```

```
...*
....
```

```
$ ./a.out
3 4
9  9  9 -1
9 -1  9  9
9  9  9  9
0 3
1 1
***3
*8**
****
```

Click here to submit to CodeCrunch.

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

```
./a.out < sweep.in | diff - sweep6.out
```

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

```
cp sweep6.c sweep7.c
```

Level 7

Name your program `sweep7.c`

Write a program that reads two integers `m` and `n` representing the dimensions of the minefield, followed by the minefield c
(representing a mine) or `-1` (safe).

The program then repeatedly reads the locations (row and column values) of any cell. If the cell is not a mine, then detern
mines surrounding this cell.

In the case where there is no surrounding mine, clear all its neighbouring cells.
Note that the above is **repeatedly applied as long as there exists cells with no surrounding mines**. The program tern
following conditions is met:

- An invalid cell location is read;
- The cell location contains a mine;
- All safe cells are cleared

Finally, the program outputs the minefield.

You may define the following functions:

`void readBoard(int board[][COL], int m, int n);` or
`void readBoard(int board[][COL], int *m, int *n);`
Read the minefield of dimensions `m` rows and `n` columns.

`void printBoard(int board[][COL], int m, int n);`
Output the minefield of dimensions `m` rows and `n` columns.

`int countNeighbour(int board[][COL], int r, int c, int m, int n);`
Count the number of mines in the neighbouring cells with respect to location `(r,c)`, while remaining within the `m-b`

`bool isValid(int r, int c, int m, int n);`
Returns `true` if location `(r,c)` is within the `m`-by-`n` minefield, or `false` otherwise.

`bool allCleared(int board[][COL], int m, int n);`
Returns `true` if all safe locations of the `m`-by-`n` minefield have been cleared, or `false` otherwise.

`void clearNeighbours(int board[][COL], int r, int c, int m, int n);`
Clears all neighbouring cells with respect to cell location `(r,c)`, while ensuring that the neighbours are within the

Tip:

- *Count the number of surrounding mines of the given cell location first.*
- *Then, go through the entire minefield and look for cell locations that have a count of zero surrounding mines and c not already cleared previously). Any time new cells are cleared, go through the minefield one more time and repea*
- *Alternatively, you can use recursion... ☺*

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

```
$ ./a.out
3 4
-1 -1 -1  9
-1 -1 -1  9
-1 -1 -1 -1
1 2
1 3
...*
..2*
....
```

```
$ ./a.out
3 4
-1 -1 -1  9
-1 -1 -1  9
-1 -1 -1 -1
1 1
-1 -1
002*
002*
001.
```

```
$ ./a.out
3 4
-1 -1 -1  9
-1 -1 -1  9
-1 -1 -1 -1
1 1
2 3
002*
002*
0011
```

```
$ ./a.out
3 4
-1 -1 -1  9
-1 -1 -1  9
-1 -1 -1 -1
0 1
4 4
002*
002*
001.
```

```
$ ./a.out
3 4
-1 -1 -1  9
-1 -1 -1  9
-1 -1 -1 -1
0 3
...*
...*
....
```

```
$ ./a.out
3 4
-1 -1 -1  9
-1 -1 -1  9
-1 -1 -1 -1
-1 -1
...*
...*
....
```

```
$ ./a.out
3 4
9  9  9 -1
9 -1  9  9
9  9  9  9
0 3
1 1
***3
*8**
****
```

Click here to submit to CodeCrunch.

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

```
./a.out < sweep.in | diff - sweep7.out
```

Submission (Course)

Select course:	CS1010E (2017/2018 Sem 1) - Programming Methodology ▼
Your Files:	

SUBMIT (only .java, .c, .cpp and .h extensions allowed)

To submit multiple files, click on the Browse button, then select one or more files. The selected file(s) will be added to the upload queue. You can repeat this step to add more files. Check that you have all the files needed for your submission. Then click on the Submit button to upload your submission.