Assignment 1 - Minesweeper

version: 1.0.2 last updated: 2020-03-14 20:30

Minesweeper

The year is 1992... Microsoft has just released Windows 3.1 and packaged with it is a beautiful game called Minesweeper. For many people, this game is the first game they will have (and the last game they will need) on their Windows computer, and it is still a classic.

If you'd like to try the game, it's available as a Google Search Game here.

In this assignment, you will be implementing COMP1511's version of this classic logic puzzle game. Our Minesweeper is a program that allows us to set up and play a game using a series of commands in a terminal. The commands are made up of integers and are typed directly into our program. Each command will make some change to a minefield, a two dimensional space that hides some mines that a player is trying not to uncover.

The aim of minesweeper is to reveal every square in the minefield except for the ones containing mines.

Minesweeper is already capable of drawing a certain view of the minefield, but it will be up to you to write code so that it can read commands and make the correct changes to the minefield. The finished product of Minesweeper is a simplified playable version of the game.

Note: At time of release of this assignment (end of Week 3), COMP1511 has not yet covered all of the techniques and topics necessary to complete this assignment. At the end of Week 3, the course has covered enough content to be able to read in a single command and process its integers, but not enough to work with two dimensional arrays like the minefield or be able to handle multiple commands ending in End-of-Input (Ctrl-D). We will be covering these topics in the lectures, tutorials, labs and a live stream in Week 4.

The Minefield

The minefield is a two dimensional array (an array of arrays) of integers that represents the space that the game is played in. We will be referring to individual elements of these arrays as squares in the minefield.

The minefield is a fixed size grid and has `SIZE` rows, and `SIZE` columns. `SIZE` is a `#define`'d constant.

Both the rows and columns start at 0, not at 1.

The top left corner of the grid is `(0, 0)` and the bottom right corner of the grid is `(SIZE - 1, SIZE - 1)`. Note that we are using rows as the first coordinate in pairs of coordinates.

For example, if we are given an input pair of coordinates `5 6`, we will use that to find a particular square in our minefield by accessing the individual element in the array: `minefield[5][6]`

		column							
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
row	**0**	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
	1	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
	2	(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
	3	(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
	4	(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
	5	(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
	6	(6,0)	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
	7	(7,0)	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

In the game of minesweeper there are states that are displayed to the player:

- A revealed square, and
- A square that has not been revealed

However, since a square that has not been revealed may or may not contain a mine, there are actually 3 values a square can take. These are represented by the following `#define`'d integers:

- `#define VISIBLE_SAFE 0`: this represents a square that has been revealed.
- `#define HIDDEN_SAFE 1`: this represents a square that has not been revealed but does not contain a mine.

- `#define HIDDEN_MINE 2`: this represents a square that has not been revealed and contains a mine.

When the program is started, all of the squares should be `HIDDEN_SAFE`. The minefield is then populated with mines (i.e., `HIDDEN_MINE`) by scanning the locations of the mines.

The way you reveal squares in the original minesweeper requires a concept not taught in COMP1511 so this has been replaced by two other revealing commands:

- `REVEAL_SQUARE`: if the selected square has adjacent mines then only reveal that square. Otherwise reveal all adjacent squares.
- `REVEAL_RADIAL`: if the selected square has adjacent mines, then only reveal that square. Otherwise, reveal lines out from the square in 45 degree increments. A line stops after a square on the edge, or a square adjacent to a mine is reached.

		column							
		0	1	2	3	4	5	6	7
row	0	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
	1	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
	2	(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
	3	(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
	4	(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
	5	(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
	6	(6,0)	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
	7	(7,0)	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

REVEAL_SQUARE

		column							
		0	1	2	3	4	5	6	7
row	0	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
	1	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
	2	(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
	3	(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
	4	(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
	5	(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
	6	(6,0)	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
	7	(7,0)	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

REVEAL_RADIAL

Please note: Adjacent refers to the 8 surrounding squares of a grid square. For example, in the diagram below, there are 8 adjacent squares (in grey) to the square in yellow.

		column							
		0	1	2	3	4	5	6	7
row	0	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
	1	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
	2	(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
	3	(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
	4	(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
	5	(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
	6	(6,0)	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
	7	(7,0)	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

Your minesweeper program will have a way of testing how many mines are in a particular row, column or square section of the minefield.

The game ends when either:

- The game is won: All of the squares are revealed except for those containing mines.
- The game is lost: A user attempts to reveal a square containing a mine.

Note: If you ever have a question in the form of "What should my program do if it is given these inputs?" you can run the minesweeper reference and copy its behaviour.

```
$ 1511 minesweeper
```

Your Task: Implementation

Your task for this assignment is to write a program that first reads in the location of the mines, then allows the user to check the number of mines in a location, and ultimately reveal the board until only the mines remain.

Your program will be given commands as a series of integers on standard input. Your program will need to scan in these integers and then make the necessary changes in the minefield.

Allowed C Features

In this assignment, there are no restrictions on C Features, except for those in the Style Guide.

We **strongly** encourage you to complete the assessment using only features taught in lectures up to and including Week 4. The only C features you will need to get full marks in the assignment are:

- `int` variables;
- `if` statements, including all relational and logical operators;
- `while` loops;
- `int` arrays, including two dimensional arrays;
- `printf` and `scanf`; and
- functions.

Using any other features will not increase your marks (and will make it more likely you make style mistakes that cost you marks).

If you choose to disregard this advice, you **must** still follow the Style Guide. You also may be unable to get help from course staff if you use features not taught in COMP1511.

Starter Code

Download the starter code (minesweeper.c) here or use this command on your CSE account to copy the file into your current directory:

```
$ cp -n /web/cs1511/20T1/activities/minesweeper/minesweeper.c .
```

`minesweeper.c` is the starting point for your minesweeper program. We've provided you with some constants and some starter code to display the minefield as basic integers on the screen; you'll be completing the rest of the program.

Input Commands

The program should first ask for the number of mines as an integer. Then, the program will scan the locations of the mines as pairs of integers in the format: `row column`

After specifying the location of the mines, each command given to the program will be a series of integers.

The first input will always be an integer representing the type of command, e.g. `2` means **How many mines in a column?**

Depending on what command the first integer specifies, you will then scan in some number of "arguments" (additional integers) that have a specific meaning for that command.

For example, `2 3` means how many mines in column 3?

Input to your program will be via standard input (similar to typing into a terminal).

You can assume that the input will always be integers and that you will always receive the correct number of arguments for a command.

Details on each command that your program must implement are shown below.

Stage 1

Stage 1 implements the ability to read in and place mines and count the number of mines in a row or column.

You can run the autotests for stage 1 by running the following command:

```
$ 1511 autotest-stage 01 minesweeper
```

Placing Mines

As is, the program currently initializes the minefield to `HIDDEN_SAFE` then prints the minefield as a grid of integers.

The program should run as follows:

1. When the program first starts, it should prompt the user: `How many mines?`
2. The program should then scan in an integer entered by the user representing the number of coordinate pairs (of the mine locations) that will be entered.
3. The program will then prompt the user to enter a list of coordinate pairs to specify the location of the mines. The coordinate pairs are entered as two integers, separated by a space, representing the row and column where a mine is located. These locations on the grid

should then contain a mine (or rather the `#define HIDDEN_MINE`).

For example, when prompted with: `Welcome to minesweeper! How many mines?` The user may enter: `2` The program will expect 2 coordinate pairs to be entered (these may or may not be valid) and prompt the user to enter these values: `Enter pairs:` The user should then enter: `1 1`, and then on the next line enter `5 7` which means place a mine at the square located at row 1, column 1, and another at row 5, column 7.

The program should then print `Game started` and print the minefield using the `print_debug_minefield` function provided in the starter code.

Invalid Input

- The first number scanned in (i.e. the number of coordinate pairs specified) will always be valid.
- If the coordinate pair specified is out of bounds of the minefield, the program should not attempt to place it on the minefield.

Examples

```
$ ./minesweeper
Welcome to minesweeper!
How many mines? 3
Enter pairs:
0 0
1 1
9 9
Game Started
2 1 1 1 1 1 1 1
1 2 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
```

Counting Mines In A Row Or Column

For the second part of Stage 1, you will be implementing the your first two commands: **Detect Row** and **Detect Column**.

Commands should be read in in a loop until one of three events: the game is won, lost or there is an EOF (Ctrl + D)

The **Detect Row** command is specified by the integer 1, followed by a row number.

The output should be given in the format: `There are n mine(s) in row r` Where n is the number of mines and r is the row number.

The **Detect Column** command is specified by the integer 2, followed by a column number.

The output should be given in the format: `There are n mine(s) in column c` Where n is the number of mines and c is the column number.

Note that after each command has been scanned in and processed the minefield is printed once again.

Invalid Input

- You can assume that all of the inputs relating to counting the number of mines in a row or column will be valid.

Examples

```
$ ./minesweeper
Welcome to minesweeper!
How many mines? 3
Enter pairs:
0 0
1 1
4 3
Game Started
2 1 1 1 1 1 1 1
1 2 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 2 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1
There are 1 mine(s) in row 1
2 1 1 1 1 1 1 1
1 2 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 2 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
2 1
There are 1 mine(s) in column 1
2 1 1 1 1 1 1 1
1 2 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 2 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
```

Stage 2

In Stage 2, you will implement a command to count the number of mines in a square section of the grid, as well as a command to reveal a 3×3 section of the grid (which makes it possible to win or lose the game).

We strongly recommend that you finish Stage 1 before attempting Stage 2, as it would be very hard to test whether Stage 2 is working without Stage 1.

Note that completing Stage 2 is not necessary to gain a passing mark in this assignment.

You can run the autotests for stage 2 by running the following command:

```
$ 1511 autotest-stage 02 minesweeper
```

Count The Number Of Mines In A Square

For the first part of Stage 2, you will be counting the number of mines in an $n \times n$ section of the grid using the **Detect Square** command.

The **Detect Square** command is specified by the number 3 followed by the row and column of the centre of the square and an odd number representing the side length: `3 row column size`

Your program should count the number of mines in this section.

Examples

```
1 1 1 1 1 1 1 1
1 2 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 2 1 1 1 1
1 1 1 1 2 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
3 3 3 3
There are 2 mine(s) in the square centered at row 3, column 3 of size 3
1 1 1 1 1 1 1 1
1 2 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 2 1 1 1 1
1 1 1 1 2 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
```

This finds the number of mines in the following section:

		column							
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
row	**0**	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
	1	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
	2	(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
	3	(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
	4	(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
	5	(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
	6	(6,0)	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
	7	(7,0)	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

Reveal Square

In order to complete the game, the user must be able to reveal the squares in the grid. For this we will create the command **Reveal Square**. This command is used to reveal a 3×3 section of the grid under certain rules. The **reveal square** command is specified by the number 4 followed by the row and column of the centre of the square: `4 row column`

Reveal square follows these rules:

- If the selected square contains a mine then the game is lost and the program should print out "Game over" and end the game. Note that the minefield should still be printed after processing the command which ends the game.
- If the selected square has adjacent mines then only reveal the selected square.
- If the selected square has no adjacent mines, then reveal all of its adjacent squares.
- If the selected square is on the edge of the minefield, only Reveal Squares which are valid squares inside the minefield.
- If at the end of revealing the squares, all of the squares except the squares containing mines have been revealed, print "Game Won!" and end the game.

Take for example, the two scenarios below. In one, the square selected to be revealed (displayed in yellow) is adjacent to a mine, so only that square is revealed. In the other scenario, there are no adjacent mines so the 8 adjacent squares are also revealed.

		column							
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
row	**0**								
	1								
	2								
	3				mine				
	4								
	5							mine	
	6								
	7								

Handling Invalid Input

- You can assume that all commands relating to **detect square** and **Reveal Square** will be valid.

Hints

- Can you create a function for the first part of this stage and use it to help you with the second part of this stage?

Examples

```
./minesweeper
Welcome to minesweeper!
How many mines? 1
Enter pairs:
3 3
Game Started
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 2 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
4 3 4
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 2 0 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
```

```
./minesweeper
Welcome to minesweeper!
How many mines? 2
Enter pairs:
3 3
4 4
Game Started
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 2 1 1 1 1
1 1 1 1 2 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
4 2 5
1 1 1 1 1 1 1 1
1 1 1 1 0 0 0 1
1 1 1 1 0 0 0 1
1 1 1 2 0 0 0 1
1 1 1 1 2 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
```

Stage 3

We strongly recommend that you finish Stage 1 and Stage 2 before attempting Stage 3.

Note that completing Stage 3 is not necessary to gain a passing mark in this assignment.

For Stage 3, you will be restricting the number of hints that can be used in a game and implementing a mode that prints the board (instead of just displaying it using `print_debug_minefield`).

You can run the autotests for stage 3 by running the following command:

```
$ 1511 autotest-stage 03 minesweeper
```

Restrict Hints

The commands **Detect Row**, **Detect Column**, and **Detect Square** are hints. In order to increase the difficulty of the game, the user should be restricted to only being able to use 3 hints. This can be any combination of **Detect Row**, **Detect Column** , and **Detect Square**.

Once the user has used their 3 hints, if they try to use another hint, they should see the message: `Help already used`

Invalid Input

- You can assume that only valid commands requesting hints will be given.

Examples

```
$ ./minesweeper
Welcome to minesweeper!
How many mines? 1
Enter pairs:
1 1
Game Started
1 1 1 1 1 1 1 1
1 2 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1
There are 1 mine(s) in row 1
1 1 1 1 1 1 1 1
1 2 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
2 1
There are 1 mine(s) in column 1
1 1 1 1 1 1 1 1
1 2 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
3 3 3 3
There are 0 mine(s) in the square centered at row 3, column 3 of size 3
1 1 1 1 1 1 1 1
1 2 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1
Help already used
1 1 1 1 1 1 1 1
1 2 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
```

Formatted Printing

The code to print out the minefield exactly as it is in the array is helpful in debugging, but it makes playing the game boring because you can see where the mines are.

We can introduce the **debug mode** and **gameplay mode** commands to switch between printing the default debug output, and printing a stylised minefield.

Debug mode is the default mode when the program is run, however, it can be switched to **gameplay mode** using the integer 5. The mode can be switched back to debug mode using the integer 6.

The format of the printed minefield is as follows:

- Non-revealed squares are represented by two hashes.
- Revealed squares contain either nothing (if there are no adjacent mines) or the number of adjacent mines if it is greater than zero.
- A smiley face is shown as the game is being played.
- A dead frowning face is shown when the game is lost.

Furthermore, when gameplay mode is activated the program should display the message: `Gameplay mode activated` and then print out the formatted minefield. For example:

```
$ ./minesweeper
Welcome to minesweeper!
How many mines? 3
Enter pairs:
0 0
1 1
2 2
Game Started
2 1 1 1 1 1 1 1
1 2 1 1 1 1 1 1
1 1 2 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
4 0 3
2 1 0 0 0 1 1 1
1 2 0 0 0 1 1 1
1 1 2 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
5
Gameplay mode activated
..
\/
    00 01 02 03 04 05 06 07
   -------------------------
00 |## ## 01       ## ## ##|
01 |## ## 02 01    ## ## ##|
02 |## ## ## ## ## ## ## ##|
03 |## ## ## ## ## ## ## ##|
04 |## ## ## ## ## ## ## ##|
05 |## ## ## ## ## ## ## ##|
06 |## ## ## ## ## ## ## ##|
07 |## ## ## ## ## ## ## ##|
   -------------------------
```

Similarly, when the program is in gameplay mode and debug mode is activated, the program should print the message `Debug mode activated` and print out the minefield in the debugging format.

```
6
Debug mode activated
2 1 0 0 0 1 1 1
1 2 0 0 0 1 1 1
1 1 2 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
```

When the game ends and the player has lost, it should reveal the locations of the mines with a pair of brackets (i.e. `()`). For example:

```
4 0 0
Game over
xx
/\
    00 01 02 03 04 05 06 07
   ------------------------
00 |() ## 01       ## ## ##|
01 |## () 02 01    ## ## ##|
02 |## ## () ## ## ## ## ##|
03 |## ## ## ## ## ## ## ##|
04 |## ## ## ## ## ## ## ##|
05 |## ## ## ## ## ## ## ##|
06 |## ## ## ## ## ## ## ##|
07 |## ## ## ## ## ## ## ##|
   ------------------------
```

Stage 4

In Stage 4, you will again be implementing a more advanced revealing method as well as a method of preventing the user from ending the game on the first click.

Again, we strongly recommend that you finish Stage 1, 2 and 3 before attempting Stage 4.

Note that completing Stage 4 is not necessary to gain a passing mark in this assignment.

You can run the autotests for stage 4 by running the following command:

```
$ 1511 autotest-stage 04 minesweeper
```

Reveal Radial

The command **Reveal Radial** is used much like the **Reveal Square** command. However, instead of revealing a 3×3 square around the selected square, an 8 pointed star-like shape is revealed outwards. **Reveal radial** is specified by the integer 7 followed by the row and column of the centre of the star: `7 row col`

Reveal radial follows the following rules:

- If the selected square contains a mine, print "Game over" and end the game. Note that the minefield should still be printed after processing the command which ends the game.
- If the selected square has adjacent mines, reveal only the selected square.
- If the selected square has no adjacent mines Reveal Radial lines from the specified square at angles of 0, 45, 90, 135, 180, 225, 270, and 315 degrees. These lines should stop at a square that has adjacent mines or a square which is on an edge.
- If all of the squares except the squares containing mines have been revealed, print "Game Won!" and end the game.

Here is an animated example of how radial expansion should work. Yellow is the selected square, grey squares have been revealed, green squares are OK to reveal, red squares you should not reveal beyond. Note that your program doesn't need to animate this reveal, this is just instructional.

		column							
		0	1	2	3	4	5	6	7
row	0					mine			
	1								
	2								
	3								
	4								
	5								
	6							mine	
	7								

Take for example, the two scenarios below. In one, the square selected to be revealed (displayed in yellow) is adjacent to a mine, so only that square is revealed. In the other scenario, there are no adjacent mines so the revealed squares expand radially.

column

		0	1	2	3	4	5	6	7
row	0								
	1								
	2								
	3				mine				
	4								
	5							mine	
	6								
	7								

Examples

```
$ ./minesweeper
Welcome to minesweeper!
How many mines? 2
Enter pairs:
1 0
5 6
Game Started
1 1 1 1 1 1 1 1
2 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 2 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
5
Gameplay mode activated
..
\/
    00 01 02 03 04 05 06 07
   -------------------------
00 |## ## ## ## ## ## ## ##|
01 |## ## ## ## ## ## ## ##|
02 |## ## ## ## ## ## ## ##|
03 |## ## ## ## ## ## ## ##|
04 |## ## ## ## ## ## ## ##|
05 |## ## ## ## ## ## ## ##|
06 |## ## ## ## ## ## ## ##|
07 |## ## ## ## ## ## ## ##|
   -------------------------
7 3 3
..
\/
    00 01 02 03 04 05 06 07
   -------------------------
00 |## ## ##    ## ##    ##|
01 |## 01 ##    ##    ## ##|
02 |## ##          ## ## ##|
03 |                       |
04 |## ##          ## ## ##|
05 |##    ##    ## 01 ## ##|
06 |   ## ##    ## ## ## ##|
07 |## ## ##    ## ## ## ##|
   -------------------------
```

Safe First Turn

In minesweeper, a user should never lose the game on the first turn. In our version of minesweeper, ensure that the user doesn't click on a mine in the first turn by shifting the grid down.

The result of this should be that all mines are now one or more squares lower than they were in the original layout. The bottom row of the grid should wrap around and be placed at the top.

It can be assumed that if a vertical line of mines occurs, then there will be no more than SIZE - 1 mines in that row or column. Your program should keep shifting the contents of the minefield downwards until the first turn becomes safe.

Examples

```
$ ./minesweeper
Welcome to minesweeper!
How many mines? 1
Enter pairs:
5 5
Game Started
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 2 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
4 5 5
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 0 1 1
1 1 1 1 1 2 1 1
1 1 1 1 1 1 1 1
```

Note: It is preferred that the first turn is interpreted as the first use of REVEAL_SQUARE or REVEAL_RADIAL, however the marking will also accept first turn interpreted as the first command that a user enters."

Hall Of Fame Challenges

If you have completed the assignment and are looking for further challenges, we have compiled a list of interesting ideas that could extend minesweeper. We call these "Hall of Fame" challenges and for anyone who completes them, we will publicise your work on the course website! You can feel free to invent your own challenges as well (and if they are at least as interesting as the ones below, we will list you on the Hall of Fame).

Before starting these, you should have completed the assignment, and be passing all autotests. If you have questions about the Hall of Fame, ask your tutor, or post a query on the course forum.

These challenges have no autotests, but completing them will result in your name being immortalised in the Hall of Fame for the assignment. They are, however, not worth any marks.

To submit a Hall of Fame attempt for testing, you should send it to cs1511.challenge@cse.unsw.edu.au with the subject line “Hall of Fame Submission z5555555”. Your email body should include a description of the challenge exercise, and a guide to how to run and test your solution. You should attach your minesweeper.c (and any other) files.

There are four challenges available:

1. Animate the explosion using sleep()- you can decide what this looks like.
2. Add colours and make the board look nicer.
3. Create a recursive reveal (replicate the one used in the original minesweeper).
4. Implement random mine placement.

These challenges are open to interpretation. If you have a question about how to complete it, ask on the course forum, or ask yourself "What would be cooler". Submitting an entry isn't a guarantee that you will make the hall of fame (but we will let you know if it does not make it, and why we have rejected it).

Style

As with all programs you write, style is an important part of your Minesweeper code. You will receive feedback on your style as part of the marking of this assessment. For an indication of how your style is going, use the `1511 style` command.

```
$ ls
minesweeper.c
$ 1511 style minesweeper.c
[style feedback]
```

Your code will be style marked by a human, so in addition to fixing issues presented by `1511 style`, you should also consider your code's readability, variable naming, and other factors that could make your code easier or harder for a human to understand.

Testing

It is important to test your code to make sure that your program can perform all the tasks necessary to become minesweeper!

There are a few different ways to test (that are described in detail below):

- Typing in your own commands. You can use the commands shown above as examples, or work out your own.
- Testing from a series of commands written in a file (example below).
- Using autotests to run through all of our test files at once.
- Running a Reference Implementation that we have created for you to determine expected output.

Testing Your Code

You can test your code by either typing commands directly into a terminal or you can type the commands into a file, and then run your program like this.

```
$ ls
minesweeper.c        test_file1.in
$ dcc -o minesweeper minesweeper.c
$ ./minesweeper < test_file1.in
[the output of running the commands in test_file1.in]
```

If you are using an input file, your file should only contain the commands which would be typed into the terminal. It should not contain any output which the program produces. For example:

```
$ cat test_file1.in
0
4 3 3
```

Automated Testing

On CSE computers (including via VLAB), the input files we have provided can all be checked at once using the command:

```
$ 1511 autotest minesweeper
```

If you wish to test only one stage at a time, the instructions for testing individual stages are shown alongside the stages themselves above.

Reference Implementation

If you have questions about what behaviour your program should exhibit, we have provided a sample solution for you to refer to.

You can use it by running the following command on CSE Computers or via VLAB:

```
$ ls
test_file1.in        test_file2.in
$ 1511 minesweeper
Welcome to minesweeper! ...
```

Assessment

Attribution of Work

This is an individual assignment. The work you submit must be your own work and only your work apart from exceptions below. Joint work is not permitted. At no point should a student read or have a copy of another student's assignment code.

You may use small amounts (< 10 lines) of general purpose code (not specific to the assignment) obtained from sites such as Stack Overflow or other publicly available resources. You should attribute clearly the source of this code in a comment with it.

You are not permitted to request help with the assignment apart from in the course forum, help sessions or from course lecturers or tutors.

Do not provide or show your assignment work to any other person (including by posting it on the forum in a public thread) apart from the teaching staff of COMP1511.

The work you submit must otherwise be entirely your own work. Submission of work partially or completely derived from any other person or jointly written with any other person is not permitted. The penalties for such an offence may include negative marks, automatic failure of the course and possibly other academic discipline. Assignment submissions will be examined both automatically and manually for such issues.

Relevant scholarship authorities will be informed if students holding scholarships are involved in an incident of plagiarism or other misconduct. If you knowingly provide or show your assignment work to another person for any reason, and work derived from it is submitted you may be penalised, even if the work was submitted without your knowledge or consent. This may apply even if your work is submitted by a third party unknown to you.

Note, you will not be penalised if your work is taken without your consent or knowledge.

Submission of Work

You are required to submit intermediate versions of your assignment.

Every time you work on the assignment and make some progress you should copy your work to your CSE account and submit it using the `give` command below.

It is fine if intermediate versions do not compile or otherwise fail submission tests.

Only the final submitted version of your assignment will be marked.

All these intermediate versions of your work will be placed in a git repo and made available to you via a web interface at this URL, replace *z5555555* with your zID:

https://gitlab.cse.unsw.edu.au/*z5555555*/20T1-comp1511-ass1_minesweeper/commits/master

This will allow you to retrieve earlier versions of your code if needed.

You submit your work like this:

```
$ give cs1511 ass1_minesweeper minesweeper.c
```

Assessment Scheme

This assignment will contribute 15% to your final mark.

80% of the marks for this assignment will be based on the performance of the code you write in `minesweeper.c`

20% of the marks for this assignment will come from hand marking of the readability of the C you have written. These marks will be awarded on the basis of clarity, commenting, elegance and style. In other words, your tutor will assess how easy it is for a human to read and understand your program.

Here is an indicative marking scheme.

HD (85-100%)	Successfully implements all of Stages 1, 2 and 3, with beautiful C code. Higher-grade High Distinctions will be awarded for the correctness and style of Stage Four.
DN (75+%)	Successfully implements all of Stages 1 and 2, with readable C code. Solutions that partially implement Stage 3 with stylistic C code will be awarded higher-grade Distinctions.
CR (65+%)	Successfully implements all of Stage 1 with readable C code. Solutions that partially implement Stage 2 will be awarded higher-grade Credits.
PS (50+%)	Reasonably readable C code that can scan in input and make some changes to the minefield as well as being able to detect mines in either a row or a column. Solutions that reject invalid input with stylistic C code will be awarded higher-grade Passes.
40-50%	A substantial attempt at part of Stage 1.
-70%	Knowingly providing your work to anyone and it is subsequently submitted (by anyone).
-70%	Submitting any other person's work. This includes joint work.

0 FL for COMP1511	Paying another person to complete work. Submitting another person's work without their consent.

The lecturer may vary the assessment scheme after inspecting the assignment submissions but it will remain broadly similar to the description above.

Due Date

This assignment is due Sunday 29 March 2020 18:00

If your assignment is submitted after this date, each hour it is late reduces the maximum mark it can achieve by 1%. For example if an assignment worth 74% was submitted 10 hours late, the late submission would have no effect. If the same assignment was submitted 30 hours late it would be awarded 70%, the maximum mark it can achieve at that time.

Change Log

Version 1.0
(2020-03-08 15:00)
- Released first assignment version.

Version 1.0.1
(2020-03-09 22:30)
- Autotest 02_detect_square_4 changed to 02_detect_square_5 to reflect assignment spec.

Version 1.0.2
(2020-03-14 20:30)
- Update reference solution and spec to clarify what the user's 'first move' is

COMP1511 20T1: Programming Fundamentals is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs1511@cse.unsw.edu.au
CRICOS Provider 00098G