THE UNIVERSITY OF SYDNEY

COMP2129 Assignment 2

Due: 11:59pm Friday, 26th April 2013

Task description

In this assignment we will be implementing a variation of the game "Minesweeper" in the C programming language. This game is commonly found in both Windows and UNIX based installations, we suggest that you play the game to gain an overall understanding of what is required by the task.

In each game, mines are hidden in a two dimensional mine field and the object of the game is to *uncover* the empty cells and *flag* the cells containing mines. To keep this assignment simple we will be implementing a simplified version of "Minesweeper" that uses standard input and output.

For example, a 4×4 mine field in our implementation will look like this:

```
+----+
|****|
|****|
|****|
|****|
+----+
```

The input of your program consists of a series of instructions that are read from standard input. In the first part, the grid size is defined along with all the specified mines. The second part involves playing the game in which cells are able to be either flagged or uncovered. The game is *won* when all mines have been flagged and all empty cells have been uncovered, or *lost* when the player uncovers a mine.

To simplify your implementation, regions without mines should *not* be automatically expanded and players are *not* allowed to unflag cells once they have been flagged. Each game will have a total of 10 mines, regardless of the grid size.

Program input

The program input consists of two sections. The first section defines the mine field and the second section describes the moves of the player. Each line of the input is one instruction. You will also be required to check for whether any invalid input is encountered.

1. The first line of the input will be of the form **`g <WIDTH> <HEIGHT>`** where **`<WIDTH>`** and **`<HEIGHT>`** are integers in the range $[1, 100]$. This defines the size of the grid to a **`<WIDTH>`** cells wide and **`<HEIGHT>`** cells high. The grid must have at least 10 cells, so this instruction is only valid if **`<WIDTH>`** $\times$ **`<HEIGHT>`** ≥ 10

2. The next 10 lines will be of the form `b <x> <y>` where `<x>` and `<y>` are integers in the range $[0,$ `<WIDTH>`$)$ and $[0,$ `<HEIGHT>`$)$ respectively. This places a mine at cell (x, y) in the grid.

3. The next lines will describe the moves of the player – either `u <x> <y>` or `f <x> <y>`.

   - A `u` instruction means that the player *uncovers* the cell (x, y)
   - A `f` instruction means that the player *flags* the cell (x, y)

   There must be exactly one player instruction for each grid cell. That is, if the grid is of size `<WIDTH>`$\times$`<HEIGHT>`, then there should be exactly `<WIDTH>` $\times$ `<HEIGHT>` `u` and `f` instructions in total. No two player instructions should refer to the same grid cell.

The input should be processed one line at a time – after reading a line of input your program should immediately act on the line of input and output described in the "Program output" section before proceeding to the next line of input.

Hint: In this assignment we recommend the use of the functions `fgets` and `sscanf` to receive input from *stdin*. This method of input is much safer and powerful, an example of their usage is shown:

```c
#define MAX_LENGTH 21

int main(void)
{
  int x, y, args;
  char command, input[MAX_LENGTH];
  fgets(input, MAX_LENGTH, stdin);
  args = sscanf(input, " %c %i %i\n", &command, &x, &y);

  return 0;
}
```

Sample input

In the following test case we construct a 4 $\times$ 4 mine field and then play the game. The player successfully flags all of the mines and uncovers the remaining cells in this game.

```
g 4 4
b 0 0
b 1 0
b 3 0
b 1 1
b 1 2
b 0 2
b 3 2
b 0 3
b 1 3
b 3 3
u 2 3
u 2 0
f 0 3
u 2 1
f 3 0
f 0 2
u 0 1
f 3 2
f 3 3
f 0 0
f 1 1
f 1 0
f 1 2
u 3 1
u 2 2
f 1 3
```

Program output

1. After reading a `g <WIDTH> <HEIGHT>` instruction, output the same instruction, e.g. output `g <WIDTH> <HEIGHT>` on a line.

2. After reading a `b <x> <y>` instruction, output the same instruction. In addition, after reading the final (10th) `b <x> <y>` line, the state of the grid should be printed (see "Outputting the grid" section for an explanation).

3. After reading a correct player instruction, output the same instruction.
   - If the player uncovers a bomb, output the message `lost` and immediately exit.
   - If every mine has been flagged and every other empty cell has been uncovered, then display the final state of the grid, output the message `won` and then exit.
   - Otherwise, the state of the grid after the instruction should be outputted.

4. If an invalid, out of place, or otherwise incorrect instruction is read at any time, `error` should be outputted and your program should exit immediately. Make sure that you do not output the instruction or the grid following an erroneous instruction.

Outputting the grid

The grid is represented using 1 character per cell and a 1 character border. A 4×4 grid looks like:

```
+----+
|****|
|****|
|****|
|****|
+----+
```

- The border consists of `-` or `|` characters, with a `+` character in each corner.
- The `*` character represents a covered cell .
- The `f` character represents a flagged cell.
- An uncovered cell is represented by a number that corresponds to the total number of bombs in the 8 adjacent cells that surround the uncovered cell.

The top left corner of the grid should have coordinates $(0, 0)$ and the bottom right corner of the grid should have coordinates (`<WIDTH>` $- 1$, `<HEIGHT>` $- 1$). The first coordinate is the x coordinate, and the second is the y coordinate. A 4×4 grid where the player has flagged $(3, 1)$ and uncovered $(0, 2)$ with 2 mines in the adjacent cells would look like:

```
+----+
|****|
|***f|
|2***|
|****|
+----+
```

Test cases

Most of the marking is done automatically, and because of this it is important that you follow the specifications of the assignment very carefully. Your program output must be in a very specific format.

To assist you with the testing of your program, a testing tool known as **`haste`** has been developed and allows you to test the correctness of your program, the coverage of your test cases and whether your program has any memory leaks. In addition, we have also included a single test case called **`sample.in`** in the **`tests/`** directory. This test case defines a basic game which is then played .

You can run the correctness, coverage and memory leak testing tools by running:

```
$ make haste
```

Writing your own test cases

We only provide you with one sample test case, but this does not test all the functionality described in assignment spec. Part of this assignment is for you to write your own test cases, based on the sample we provide. This is to ensure that your program will work correctly for any valid or invalid input.

You should place all of your test cases in the **`tests/`** directory. Each testcase should be in the form of *sample.in*. We recommend that the names of your test cases are descriptive so that you know what each is testing, e.g. **`too-few-mines.in`** and **`incomplete-game.in`**

We will also be assessing the *coverage* of your test cases, this refers to the percentage of code that the execution of your code passes through. Coverage is an important metric in testing because lines of code that are not covered by a set of test cases are (by definition) never tested. This makes them susceptible to bugs that are difficult to detect. Cases where your coverage may be incomplete include:

- A grid with *less* than 10 mines is never tested, and the corresponding **`if branch`** is missed.
- An *incomplete* game is never tested, where there are less **`u`** and **`f`** instructions than cells.

In both of these cases, there could be bugs that may still exist in your code even though all of your other tests pass, as you are not testing all the possible branches of the prototype.

To achieve 100% coverage, you will need to carefuly read the assignment specification and come with test cases that check every single aspect of the task – including valid, invalid and incorrect inputs.

Hint: You can use the included Makefile to check the correctness of your program using your own test cases, and also report on the overall coverage that you have against the staff prototype. See "The Makefile" section for more details.

The Makefile

We have provided you with a Makefile that can be used to compile and test your code using the test cases you define. The included Makefile will only run correctly on the **`ucpu0`** and **`ucpu1`** machines.

- **`$ make`**
  will compile your **`minesweeper.c`** file into an executable file **`./minesweeper`**
- **`$ make test`**
  will compile your program and then check the correctness of your program using the test cases that you define in your **`tests/`** directory and highlight any differences between the **`expected`** output of the staff prototype and the **`observed`** output of your program.
- **`$ make cover`**
  will determine the **`coverage`** obtained using all of the test cases defined in your **`tests/`** directory against the staff prototype, and also output some hints on what is not being covered.
- **`$ make rain`**
  will compile your program and then check whether your program has any memory leaks using the test cases that you define in your **`tests/`** directory. If any test cases require further investigation then run valgrind on each test, e.g. **`$ valgrind ./minesweeper < sample.in`**

Marking

5 **marks** are assigned based on automatic tests for the *correctness* of your program.
This component will use our own (hidden) test cases that cover every aspect of the specification. Note that your test cases will also be assessed and used in this component.

3 **marks** are assigned based on automatic tests for your test case *coverage* of the specification.
The minimum required coverage is 70%, every percent above 70% will attain 0.1 marks.

2 **marks** are assigned based on a manual inspection of the style and quality of your code.

Warning: Any attempts to deceive the automatic marking will result in an immediate 0 for the entire assignment. Negative marks can also be assigned from the manually marked component if your code is unnecessarily and/or deliberately obfuscated.