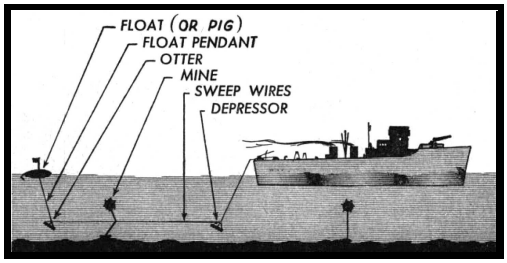
MSc (Computing Science) 2015-2016
C/C++ Laboratory Examination

Imperial College London

Tuesday 12 January 2016, 11h00 – 13h10



- ☞ You are advised to use the first 10 minutes for reading time.
- ☞ You must complete and submit a working program by 13h10.
- ☞ Log into the Lexis exam system using your DoC login as both your login and as your password (**do not use your usual password**).
- ☞ You must add to the pre-supplied header file **minesweeper.h**, pre-supplied implementation file **minesweeper.cpp** and must create a **makefile** according to the specifications overleaf.
- ☞ You will find source files **minesweeper.cpp**, **minesweeper.h** and **main.cpp**, and data files **mines.dat**, **partial.dat**, **solution.dat** and **bonus.dat** in your Lexis home directory (**/exam**). If one of these files is missing alert the invigilators.
- ☞ **Save your work regularly.**
- ☞ Please log out once the exam has finished. No further action needs to be taken to submit your files.
- ☞ No communication with any other student or with any other computer is permitted.
- ☞ You are not allowed to leave the lab during the first 15 minutes or the last 10 minutes.
- ☞ **This question paper consists of 8 pages.**

Image Credit: U.S. Navy All Hands magazine November 1952, p. 4.

Problem Description

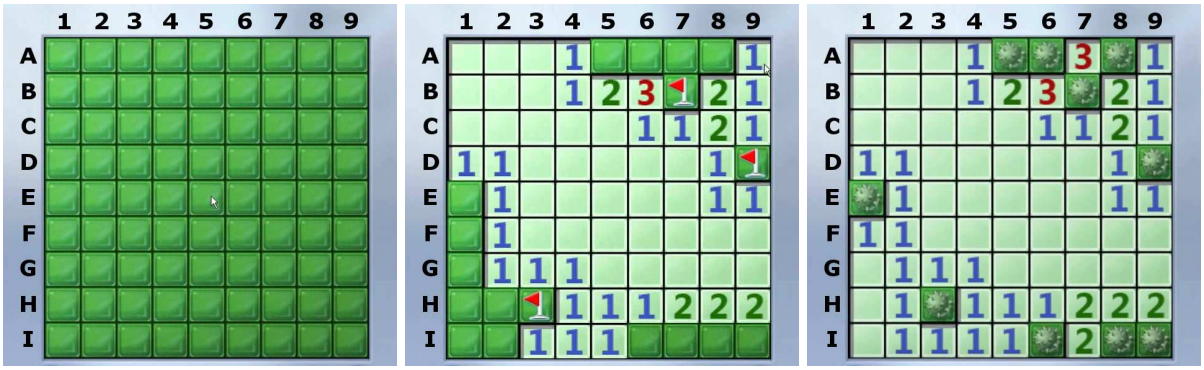


Figure 1: A minesweeper game beginning (left), in progress (middle), and completed (right)

Minesweeper is a single-player game played on a rectangular grid (here assumed to be 9×9). As shown on the left in Figure 1, the player is initially presented with a grid of covered squares. Some of the covered squares conceal mines, and it is the player's goal to work out exactly where they are without exploding any of them.

At each turn, the player must chose to either *uncover* a square or to *flag* a square as containing a mine. If a player uncovers a square containing a mine, the mine explodes and the player loses the game. Otherwise a digit is displayed in the uncovered square which indicates how many adjacent (neighbouring) squares contain mines. *If there are no adjacent mines, the square becomes blank and all adjacent squares are recursively uncovered.* If all mine-free squares become uncovered, the player wins the game (as shown on the right in Figure 1).

A player can use the digits displayed on squares to deduce that one or more squares definitely contains a mine (and should be flagged). For example, in the middle panel of Figure 1, it can be deduced that squares A5 and A6 contain mines since otherwise B5 would not have the value 2. Equally, a player may deduce that it is safe to uncover one or more squares. For example, because H3 is flagged and G2 has the value 1, it must be safe to uncover F1, G1, H1 and H2.

Your challenge is to write functions which will be useful for playing a Minesweeper game. To aid you in developing your program, you are supplied with test boards related to Figure 1.

Pre-supplied functions and files

You are supplied with a main program in **main.cpp**, and with a **mines.dat** data file representing the locations of mines in Figure 1. You are also supplied with the partially complete playing board shown in the middle of Figure 1 in **partial.dat** and the solved playing board shown on the right of Figure 1 in **solution.dat**[1].

You can use the UNIX command **cat** to inspect these files, e.g.

```
% cat mines.dat    % cat partial.dat  % cat solution.dat
....**.*.             1????1             1**3*1
......*..             123*21             123*21
.........               1121               1121
........*          11     1*          11     1*
*........          ?1     11          *1     11
.........          ?1                 11
.........          ?111                111
..*......          ??*111222           1*111222
.....*.**          ??111????           1111*2**
```

We will store the locations of the mines in a two-dimensional (9×9) array of characters called `mines`, while the playing board shown to the user will be stored in a two-dimensional array called `revealed`.

You are supplied with some helper functions (with prototypes in **minesweeper.h** and implementations in the file **minesweeper.cpp**):

- `void load_board(const char *filename, char board[9][9])` reads in characters from the file with name `filename` into the two-dimensional character array `board`.
- `void display_board(char board[9][9])` displays a 2D character array `board`. Row indices (in the form of letters 'A' to 'I') and column indices (in the form of digits '1' to '9') are included in the output to help with the identification of particular board positions.
- `void initialise_board(char board[9][9])` initialises a playing board with '?' characters to indicate that all squares are initially covered.

[1]Note **partial.dat** and **solution.dat** are provided for your reference only, i.e. you should *not* assume their general availability or use their contents to compute the solution.

To illustrate the use of the above functions, consider the code:

```
  char mines[9][9], revealed[9][9];
  load_board("mines.dat", mines))
  display_board(mines);
  cout << "Calling initialise_board()..." << endl;
  initialise_board(revealed);
  display_board(revealed);
```

This results in the output:

```
Loading board from file 'mines.dat'... Success!
  123456789
 +---------+
A|    ** * |
B|      *  |
C|         |
D|        *|
E|*        |
F|         |
G|         |
H|  *      |
I|     * **|
 +---------+

Calling initialise_board()...
  123456789
 +---------+
A|?????????|
B|?????????|
C|?????????|
D|?????????|
E|?????????|
F|?????????|
G|?????????|
H|?????????|
I|?????????|
 +---------+
```

Specific Tasks

1. Write a Boolean function `is_complete(mines, revealed)` which takes two 9×9 arrays of characters, the first (`mines`) representing mine locations and the second (`revealed`) the current playing board, and returns true if and only if all non-mine squares in the playing board have been uncovered. For example, the code:

   ```
   load_board("mines.dat", mines);
   load_board("solution.dat", revealed);
   cout << "Game is ";
   if (!is_complete(mines, revealed))
     cout << "NOT ";
   cout << "complete." << endl;
   ```

   should display the output

   ```
   Loading board from file 'mines.dat'... Success!
   Loading board from file 'solution.dat'... Success!
   Game is complete.
   ```

2. Write a function `count_mines(position, mines)` which returns the number of mines around a particular square. Here `position` is a two-character string denoting row and column board coordinates (e.g. "I8") and `mines` is a 2D character array of mine locations. For example, the code:

   ```
   load_board("mines.dat", mines);
   cout << "Found " << count_mines("A7", mines)
        << " mine(s) around square 'A7'" << endl;
   cout << "Found " << count_mines("E5", mines)
         << " mine(s) around square 'E5'" << endl;
   cout << "Found " << count_mines("H9", mines)
        << " mine(s) around square 'H9'" << endl;
   ```

   should display the output

   ```
   Loading board from file 'mines.dat'... Success!
   Found 3 mine(s) around square 'A7'
   Found 0 mine(s) around square 'E5'
   Found 2 mine(s) around square 'H9'
   ```

3. Write a function `make_move(position, mines, revealed)` which uncovers or flags a square. Here `mines` is an input parameter representing mine locations while `revealed` is an input/output parameter presenting the current playing board. The first two characters of `position` denote the row and column board coordinates. If no other characters are present in `position`, the move is to *uncover* the square in `revealed`[2]. If a `'*'` is present as the third character, the move is to *flag* the square in `revealed`. The return value should be of type `MoveResult` (see **minesweeper.h**):

```
enum MoveResult { INVALID_MOVE=-3, REDUNDANT_MOVE=-2,
  BLOWN_UP=-1, SOLVED_BOARD=1, VALID_MOVE=0 };
```

`INVALID_MOVE` is returned when the row and column coordinates are not valid or if the third character is present, but is not a `'*'`. `REDUNDANT_MOVE` is returned if the target position corresponds to a square that has already been uncovered or flagged. `BLOWN_UP` is returned if the move uncovers a mine. `SOLVED_BOARD` is returned if all non-mine squares in `revealed` have been uncovered. Otherwise, `VALID_MOVE` is returned. For example, the calls:

```
load_board("mines.dat", mines);
initialise_board(revealed);
MoveResult result1 = make_move("B6", mines, revealed);
MoveResult result2 = make_move("E5", mines, revealed);
MoveResult result3 = make_move("H3*", mines, revealed);
```

should result in `revealed` successively taking on the values:

```
  123456789          123456789          123456789
 +---------+        +---------+        +---------+
A|?????????|       A|   1?????|       A|   1?????|
B|?????3???|       B|   123???|       B|   123???|
C|?????????|       C|     112?|       C|     112?|
D|?????????|       D|11     1?|       D|11     1?|
E|?????????|       E|?1     11|       E|?1     11|
F|?????????|       F|?1       |       F|?1       |
G|?????????|       G|?111     |       G|?111     |
H|?????????|       H|???111222|       H|??*111222|
I|?????????|       I|?????????|       I|?????????|
 +---------+        +---------+        +---------+
```

with `result1`, `result2` and `result3` all set to `VALID_MOVE`.

[2]Recall the rule that if there are no adjacent mines, the square becomes blank and all adjacent squares are recursively uncovered.

4. Write Boolean function `find_safe_move(revealed, move)` which determines if a risk-free move[3] is available starting from the current playing board `revealed`. The return value of the function should be `true` if a risk-free move is available, in which case output string `move` should contain the move. Otherwise the return value of the function should be `false` and the output string `move` should be the empty string.

   For example, the code:

```
load_board("mines.dat", mines);
load_board("partial.dat", revealed);
display_board(revealed);

cout << "Safe move sequence: " << endl;
char move[512];
while (find_safe_move(revealed, move)) {
  cout << move << " ";
  make_move(move, mines, revealed);
}
cout << endl;
```

   should result in output similar to:

```
Loading board from file 'partial.dat'... Success!
  123456789
 +---------+
A|   1????1|
B|   123*21|
C|     1121|
D|11     1*|
E|?1     11|
F|?1       |
G|?111     |
H|??*111222|
I|??111????|
 +---------+
Safe move sequence:
A5* A8* A6* A7 E1* F1 G1 I6* I7 I8* I9*
```

(The four parts carry, resp., 15%, 20%, 40% and 25% of the marks)

[3] That is, an *uncover* or *flag* move which can be safely made without any guesswork.

What to hand in

Place your function implementations in the file **minesweeper.cpp** and corresponding function declarations in the file **minesweeper.h**. Use the file **main.cpp** to test your functions. Create a **makefile** which will compile your submission into an executable file entitled **minesweeper**.

Hints

1. You will save time if you begin by studying the main program in **main.cpp**, the header file **minesweeper.h**, the pre-supplied functions in **minesweeper.cpp** and the given data files.
2. All the questions will be **much** easier if you exploit the pre-supplied helper functions.
3. Feel free to define any of your own helper functions which would help to make your code more elegant.
4. Try to attempt all questions. If you cannot get one of the questions to work, try the next one.
5. You are not explicitly required to use recursion in your answer to the questions. However, you may make use of recursion wherever you feel it would make your solution more elegant (especially in Question 3).

Bonus Challenge

Write a function `solve_board(mines,revealed,moves)` which works out the set of *risky* moves[4] required to solve a minesweeper puzzle and outputs a string containing these via output parameter `moves`. You can assume safe (risk-free) moves will be made whenever possible (in between the risky ones). When choosing between competing risky moves, you should select the one which will (when taken together with all the safe moves it makes possible) uncover the most squares. Since the **mines.dat** example is not very interesting in this respect (you should find only one risky move is required e.g. `A1`), you are provided with a **bonus.dat** file which has a non-trivial solution.

[4]That is, "unsafe" *uncover* moves which require an element of guesswork on behalf of the player.