CS121

Computer Programming I

FINAL PROJECT REPORT

Minesweeper

Contents:

Team Members:

- Abdelhakeem Osama [34]
- Abdelrahman Shams [40]

Description

This is a console-based implementation of the famous classic puzzle video game, Minesweeper.

As with any Minesweeper, the player is supposed to open all cells of a rectangular board containing mines, without detonating any of them (i.e.: without opening any cell that contains a mine.)

The first cell the player opens always has no mine. Cells that have no mines but are neighboring to cells with mines contain the number of adjacent cells with mines. This way the player can more easily guess where the mines actually are.

Along the way, the player can “flag” some cells as having mines to avoid opening them. Flagged cells can also make gameplay faster. For instance, the player can choose to open an open cell containing a certain number, in which case if the cell is surrounded by an equal number of flags then all neighboring cells are opened except those with flags (whether those neighboring cells contain mines or not), all in one move!

In addition, the player can mark any cell with a question mark, when unsure whether it contains a mine or not, to avoid opening it later. The player can unmark any cell previously marked with a flag or a question mark.

The player loses when a cell with a mine is opened. Otherwise, the player does not win unless all cells with no mines are opened. The player does not necessarily have to flag cells which contain mines.

The main features of our implementation include:

- Rectangular grid with custom dimensions, from 2x2 up to 30x30
- At any time, the game can be saved to a file which can be loaded later to continue the game from the same point
- Sorted table of winners with their names and scores

Design Overview

1. Input Module

Handles validated input of various data types from the user.

2. Grid Module

Handles grid initialization and display.

3. Game Actions Module

Handles game actions, including:

- Opening a cell.
- Flagging a cell.
- Marking a cell with a question mark.
- Unmarking a cell.

4. Game Loop Module

Handles the main game loop till the game ends in either state: win or lose.

5. Saving Module

Handles saving and loading game state.

6. Scores Module

Handles storage and retrieval of scores of winners.

Data Structures

Cell	
Member	**Description**
`char status`	Represent the status of the cell 0: Closed 1: Open 2: Flagged 3: Marked
`char type`	Type of the cell ‘0’: Empty ‘V’: Empty visited cell (all adjacent cells are open) ‘1’~’8’: Number of adjacent cells with mines ‘*’: Mine ‘!’: Loss mine (mine that made the player lose) ‘M’: Missed mine ‘-’: Incorrect flag

Position	
Member	**Description**
`int x`	Row Number
`int y`	Column Number

ScoreEntry	
Member	**Description**
`char playerName[128]`	Player Name
`int score`	Score

Functions

1. Input Module

- `void flushstdin(void);`
  Reads from stdin until an ENTER character is encountered (which is also read).
  Also used to pause the game until the player presses ENTER.
- `void trimSpaces(char *str);`
  Trims leading and trailing spaces from a string.
- `char ab_getchar(void);`
- `int getint(void);`
- `void getstring(int size, char *str);`
  NOTE: Two characters at the end of the string are reserved for '\n' and '\0', so we actually read (size-2) characters.

2. Grid Module

- `Position getRandomPos(int m, int n);`
  Returns a random position in a grid of (m X n) dimensions.
  NOTE: for this function to work, srand(time(NULL)) must be called at least once before using this function or any of its dependents to seed the random number generator.
- `char positionsEqual(Position a, Position b);`
  Returns 1 if Positions (a) and (b) have the same coordinates, 0 otherwise.
- `char arrayContains(Position *arr, int arrsize, Position key);`
  Returns 1 if (arr) contains the position (key), 0 otherwise.
- `void prepareGrid(int m, int n, Cell grid[m][n]);`
  Zeroes out (empties) all cells and sets them closed.
- `void initGrid(Position firstPos, int m, int n, Cell grid[m][n]);`
  Populates a grid of (m X n) dimensions with mines and numbers. (call prepareGrid first!)
  This function shall be called after the player makes their first move whose position (firstPos) must not contain a mine
  NOTE: Prior validation for (m) and (n) is required!
- `void printGrid(int m, int n, Cell grid[m][n], char *filename, char debug);`
  Prints a grid of (m X n) dimensions.
  To print to (stdout), simply pass an empty string for (filename)
  For debugging mode (i.e.: all cells open), pass a non-zero value for (debug)

3. Game Actions Module

- `void openEmpty(int m, int n, Cell grid[m][n], Position cellPos);`
  Opens an empty cell and all adjacent empty cells recursively.
- `char openCell(int m, int n, Cell grid[m][n], Position cellPos);`
  Opens the cell at (cellPos)
  If the cell is already open, there are two cases:
  1) If the cell contains a number, say (n), and (n) adjacent cells are marked with flags, then all adjacent cells without flags are opened, whether they contain mines or not.
  2) Otherwise, the function returns with error code 1
  If the opened cell is empty, all empty adjacent cells are opened.
  The value of any opened cell that contains a mine is replaced by an exclamation mark.
  Returns 0 on success. Otherwise, an error code is returned as follows:
  1: The cell is already open.
  2: One or more of the opened cells contain mines.
  3: The cell is flagged.
  4: The cell is marked with a question mark.
- `char flagCell(int m, int n, Cell grid[m][n], Position cellPos);`
  Flags the cell at (cellPos)
  Returns 0 on success. Otherwise, an error code is returned as follows:
  1: Cell is open.
  2: Cell is already flagged.
  3: Cell is marked with a question mark.
- `char markCell(int m, int n, Cell grid[m][n], Position cellPos);`
  Marks the cell at (cellPos) with a question mark.
  Returns 0 on success. Otherwise, an error code is returned as follows:
  1: Cell is open.
  2: Cell is flagged.
  3: Cell is already marked with a question mark.
- `char unmarkCell(int m, int n, Cell grid[m][n], Position cellPos);`
  Unmarks the cell at (cellPos)
  Returns an error code as follows:
  1: Flag removed successfully.
  2: Question mark removed successfully.
  3: The cell is not flagged nor marked.

4. Game Loop Module

- `long long calculateScore(int m, int n, int timePassed, int movesDone);`
  Calculates the score of the player after (timePassed) seconds have passed and (movesDone) moves have been done.
- `void printHeader(long long timePassed, int movesDone, int flagged, int marked, long long score);`
  Prints a header line showing the time passed in seconds, number of moves done, and current score, in addition to the number of flagged cells and number of cells marked with a question mark.
- `void getMove(int m, int n, Position *pos);`
  Gets the coordinates of the next move from player.
- `char hasWon(int m, int n, Cell grid[m][n]);`
  Checks the grid to determine if the player has won.
  Returns 1 if player has won, 0 otherwise
- `char startGame(int m, int n, Cell grid[m][n], int timePassed, int movesDone);`
  Starts the game with the following initial conditions:
  (timePassed): Time passed in seconds since the start of the current game
  (0 in case of starting a new game)
  (movesDone): Number of moves done since the start of the current game
  (0 in case of starting a new game)
  The player wins when all the cells that have no mines are opened.
  In case the player loses, incorrect flags should be replaced with '-' and missed mines should be replaced with 'M'
  Returns 0 on winning. Otherwise, an error code is returned as follows:
  1: Player lost.
  2: Player requested to save and exit to main menu.
  3: Player requested to exit to main menu without saving.

5. Saving Module

- `char saveGame(int m, int n, Cell grid[m][n], int timePassed, int movesDone, char *fileName);`
  Saves the status and grid of the current game to a file with the following format:

4 bytes:	**'ABMS' (for AB Team Minesweeper)**
int:	**Time passed since the start of the current game.**
int:	**Moves done since the start of the current game.**
int:	**Number of rows.**
int:	**Number of columns.**
(2*rows*columns) bytes:	**Grid array (elements are of Cell data structure)**

  Returns 0 on success, 1 otherwise.
- `char loadGame_info(int *m, int *n, int *timePassed, int *movesDone, char *fileName);`
  Loads the status of a previously saved game from a file into the variables pointed to by (m), (n), (timePassed), (movesDone)
  Returns 0 on success, 1 otherwise.
- `char loadGame_grid(int m, int n, Cell grid[m][n], char *fileName);`
  Loads the grid of a previously saved game from a file, given that its dimensions are (m X n)
  Returns 0 on success, 1 otherwise.

6. Scores Module

- `char strEqual(char *s1, char *s2);`
  Returns 1 if (s1) is identical to (s2), ignoring letter case. Otherwise, returns 0
- `char readScores(char *fileName, int numEntries, ScoreEntry *scores);`
  Reads table of scores from file and stores it in (scores)
  Return 1 on success, 0 otherwise
- `char addScore(char playerName[128], long long score, char *fileName);`
  Adds the score of the player to the table of scores file.
  Returns 1 on success, 0 otherwise.
- `char displayScores(char *fileName);`
  Reads table of scores from file.
  Returns 1 on success, 0 otherwise.

Scores File Format

4 bytes:	**'ABMS'**
int:	**Number of score entries**
int:	**Length of player name (n)**
(n) bytes:	**Player name**
long long:	**Player score**

.
.
.