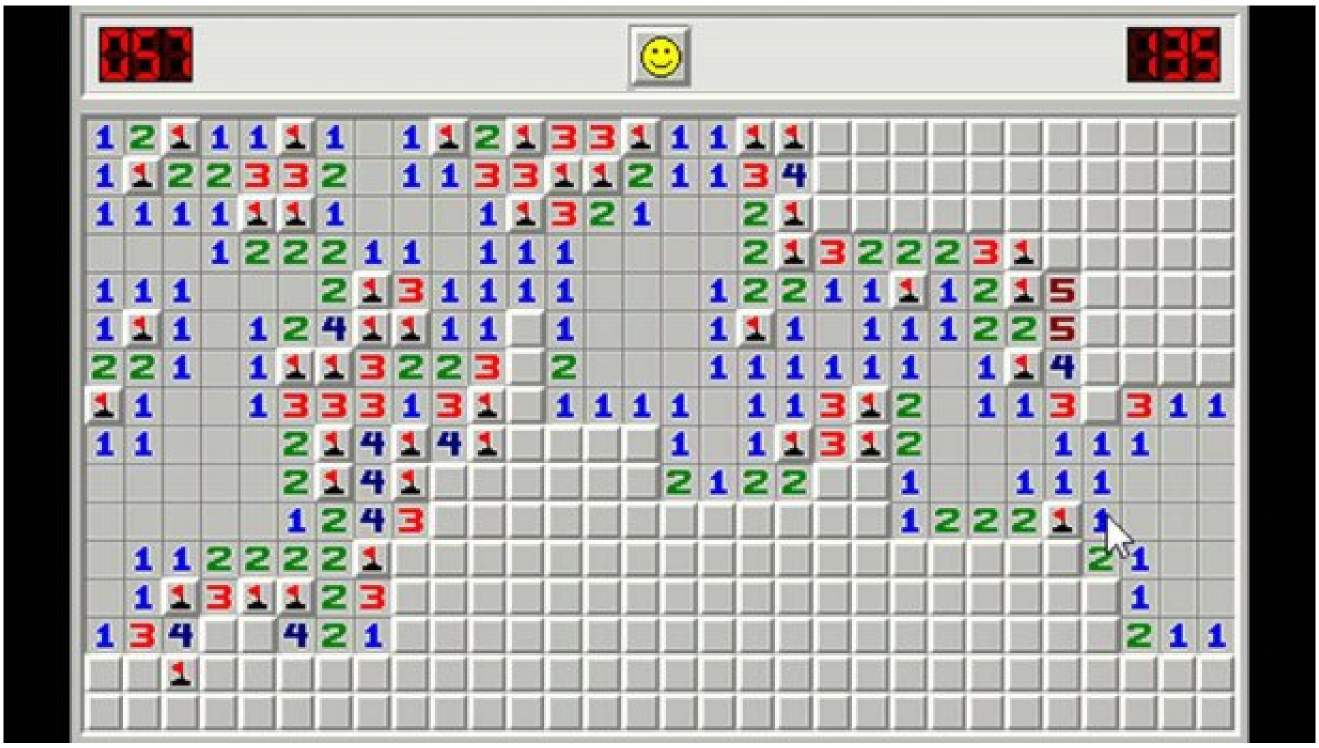
Implementation of Minesweeper Game

Remember the old Minesweeper ?

We play on a square board and we have to click on the board on the cells which do not have a mine. And obviously we don't know where mines are. If a cell where a mine is present is clicked then we lose, else we are still in the game.

There are three levels for this game-

1. **Beginner** – 9 * 9 Board and 10 Mines
2. **Intermediate** – 16 * 16 Board and 40 Mines
3. **Advanced** – 24 * 24 Board and 99 Mines



Probability of finding a mine –

- Beginner level – 10/81 (0.12)
- Intermediate level – 40/256 (0.15)
- Advanced level – 99 / 576 (0.17)

The increasing number of tiles raises the difficulty bar. So the complexity level increases as we proceed to next levels.

It might seem like a complete luck-based game (you are lucky if you don’t step over any mine over the whole game and unlucky if you have stepped over one). But this is not a complete luck based game. Instead you can win almost every time if you follow the hints given by the game itself.

Hints for Winning the Game

- When we click on a cell having adjacent mines in one or more of the surrounding eight cells, then we get to know how many adjacent cells have mines in them. So we can do some logical guesses to figure out which cells have mines.
- If you click on a cell having no adjacent mines (in any of the surrounding eight cells) then all the adjacent cells are automatically cleared, thus saving our time.
- So we can see that we don’t always have to click on all the cells not having the mines (total number of cells – number of mines) to win. If we are lucky then we can win in very short time by clicking on the cells which don’t have any adjacent cells having mines.

Implementation

Two implementations of the game are given here:

1. In the first implementation, the user’s move is selected randomly using rand() function.
2. In the second implementation, the user himself select his moves using scanf() function.

Also there are two boards- **realBoard** and **myBoard**. We play our game in **myBoard** and **realBoard** stores the location of the mines. Throughout the game, **realBoard** remains unchanged whereas **myBoard** sees many changes according to the user’s move.

We can choose any level among – BEGINNER, INTERMEDIATE and ADVANCED. This is done by passing one of the above in the function – **chooseDifficultyLevel()** [However in the user-input game this option is asked to the user before playing the game].

Once the level is chosen, the **realBoard** and **myBoard** are initialized accordingly and we place the mines in the **realBoard** randomly. We also assign the moves using the function **assignMoves()** before playing the game [However in the user-input game the user himself assign the moves during the whole game till the game ends].

We can cheat before playing (by knowing the positions of the mines) using the function – **cheatMinesweepeer()**. In the code this function is commented . So if you are afraid of losing then uncomment this function and then play !

Then the game is played till the user either wins (when the user never steps/clicks on a mine-containing cell) or lose (when the user steps/clicks on a mine-containing cell). This is represented in a while() loop. The while() loop terminates when the user either wins or lose.

The **makeMove()** function inside the while loop gets a move randomly from then randomly assigned moves. [However in the user-input game this function prompts the user to enter his own move].

Also to guarantee that the first move of the user is always safe (because the user can lose in the first step itself by stepping/clicking on a cell having a mine, and this would be very much unfair), we put a check by using the if statement – if (**currentMoveIndex** == 0)

The lifeline of this program is the recursive function – **playMinesweeperUtil()**

This function returns a true if the user steps/clicks on a mine and hence he loses else if he step/click on a safe cell, then we get the count of mines surrounding that cell. We use the function **countAdjacentMines()** to calculate the adjacent mines. Since there can be maximum 8 surrounding cells, so we check for all 8 surrounding cells.

If there are no adjacent mines to this cell, then we recursively click/step on all the safe adjacent cells (hence reducing the time of the game-play). And if there is atleast a single adjacent mine to this cell then that count is displayed on the current cell. This is given as a hint to the player so that he can avoid stepping/clicking on the cells having mines by logic.
Also if you click on a cell having no adjacent mines (in any of the surrounding eight cells) then all the adjacent cells are automatically cleared, thus saving our time.

So we can see that we don't always have to click on all the cells not having the mines (total number of cells – number of mines) to win. If we are lucky then we can win in very short time by clicking on the cells which don't have any adjacent cells having mines.

The user keeps on playing until he steps/clicks on a cell having a mine (in this case the user loses) or if he had clicked/stepped on all the safe cell (in this case the user wins).

```
// A C++ Program to Implement and Play Minesweeper

#include<bits/stdc++.h>
using namespace std;

#define BEGINNER 0
#define INTERMEDIATE 1
#define ADVANCED 2
#define MAXSIDE 25
#define MAXMINES 99
#define MOVESIZE 526 // (25 * 25 - 99)

int SIDE ; // side length of the board
int MINES ; // number of mines on the board

// A Utility Function to check whether given cell (row, col)
// is a valid cell or not
bool isValid(int row, int col)
{
    // Returns true if row number and column number
    // is in range
    return (row >= 0) && (row < SIDE) &&
           (col >= 0) && (col < SIDE);
}

// A Utility Function to check whether given cell (row, col)
// has a mine or not.
bool isMine (int row, int col, char board[][MAXSIDE])
{
    if (board[row][col] == '*')
        return (true);
    else
        return (false);
}

// A Function to get the user's move
void makeMove(int *x, int *y)
```

```c
{
    // Take the input move
    printf("Enter your move, (row, column) -> ");
    scanf("%d %d", x, y);
    return;
}

// A Function to print the current gameplay board
void printBoard(char myBoard[][MAXSIDE])
{
    int i, j;

    printf ("    ");

    for (i=0; i<SIDE; i++)
        printf ("%d ", i);

    printf ("\n\n");

    for (i=0; i<SIDE; i++)
    {
        printf ("%d   ", i);

        for (j=0; j<SIDE; j++)
            printf ("%c ", myBoard[i][j]);
        printf ("\n");
    }
    return;
}

// A Function to count the number of
// mines in the adjacent cells
int countAdjacentMines(int row, int col, int mines[][2],
                      char realBoard[][MAXSIDE])
{

    int i;
    int count = 0;

    /*
        Count all the mines in the 8 adjacent
        cells

            N.W   N   N.E
              \   |   /
               \  |  /
            W----Cell----E
                 / | \
               /   |  \
            S.W    S   S.E

        Cell-->Current Cell (row, col)
        N -->  North        (row-1, col)
        S -->  South        (row+1, col)
        E -->  East         (row, col+1)
        W -->  West            (row, col-1)
        N.E--> North-East   (row-1, col+1)
        N.W--> North-West   (row-1, col-1)
        S.E--> South-East   (row+1, col+1)
        S.W--> South-West   (row+1, col-1)
    */

    //----------- 1st Neighbour (North) ------------

        // Only process this cell if this is a valid one
        if (isValid (row-1, col) == true)
        {
               if (isMine (row-1, col, realBoard) == true)
               count++;
        }
    //----------- 2nd Neighbour (South) ------------

        // Only process this cell if this is a valid one
        if (isValid (row+1, col) == true)
        {
               if (isMine (row+1, col, realBoard) == true)
               count++;
        }

    //----------- 3rd Neighbour (East) ------------

        // Only process this cell if this is a valid one
        if (isValid (row, col+1) == true)
        {
            if (isMine (row, col+1, realBoard) == true)
               count++;
        }

    //----------- 4th Neighbour (West) ------------

        // Only process this cell if this is a valid one
        if (isValid (row, col-1) == true)
        {
               if (isMine (row, col-1, realBoard) == true)
               count++;
        }

    //----------- 5th Neighbour (North-East) ------------

        // Only process this cell if this is a valid one
        if (isValid (row-1, col+1) == true)
        {
            if (isMine (row-1, col+1, realBoard) == true)
               count++;
        }

     //----------- 6th Neighbour (North-West) ------------

        // Only process this cell if this is a valid one
        if (isValid (row-1, col-1) == true)
        {
             if (isMine (row-1, col-1, realBoard) == true)
               count++;
        }

     //----------- 7th Neighbour (South-East) ------------

        // Only process this cell if this is a valid one
        if (isValid (row+1, col+1) == true)
        {
               if (isMine (row+1, col+1, realBoard) == true)
               count++;
        }

    //----------- 8th Neighbour (South-West) ------------

        // Only process this cell if this is a valid one
        if (isValid (row+1, col-1) == true)
        {
            if (isMine (row+1, col-1, realBoard) == true)
               count++;
        }

    return (count);
}

// A Recursive Fucntion to play the Minesweeper Game
bool playMinesweeperUtil(char myBoard[][MAXSIDE], char realBoard[][MAXSIDE],
            int mines[][2], int row, int col, int *movesLeft)
```

```
{

    // Base Case of Recursion
    if (myBoard[row][col] != '-')
        return (false);

    int i, j;

    // You opened a mine
    // You are going to lose
    if (realBoard[row][col] == '*')
    {
        myBoard[row][col]='*';

        for (i=0; i<MINES; i++)
            myBoard[mines[i][0]][mines[i][1]]='*';

        printBoard (myBoard);
        printf ("\nYou lost!\n");
        return (true) ;
    }

    else
     {
        // Calculate the number of adjacent mines and put it
        // on the board
        int count = countAdjacentMines(row, col, mines, realBoard);
        (*movesLeft)--;

        myBoard[row][col] = count + '0';

        if (!count)
        {
            /*
            Recur for all 8 adjacent cells

                N.W   N   N.E
                  \   |   /
                      \  |  /
                W----Cell----E
                     / | \
                   /   |  \
                S.W    S   S.E

            Cell-->Current Cell (row, col)
            N -->  North        (row-1, col)
            S -->  South        (row+1, col)
            E -->  East         (row, col+1)
            W -->  West            (row, col-1)
            N.E--> North-East   (row-1, col+1)
            N.W--> North-West   (row-1, col-1)
            S.E--> South-East   (row+1, col+1)
            S.W--> South-West   (row+1, col-1)
            */

                //----------- 1st Neighbour (North) ------------

            // Only process this cell if this is a valid one
            if (isValid (row-1, col) == true)
            {
                   if (isMine (row-1, col, realBoard) == false)
                   playMinesweeperUtil(myBoard, realBoard, mines, row-1, col, mo
            }

            //----------- 2nd Neighbour (South) ------------

            // Only process this cell if this is a valid one
            if (isValid (row+1, col) == true)
            {
                   if (isMine (row+1, col, realBoard) == false)
```

```
                    playMinesweeperUtil(myBoard, realBoard, mines, row+1, col, m
            }

            //----------- 3rd Neighbour (East) ------------

            // Only process this cell if this is a valid one
            if (isValid (row, col+1) == true)
            {
                if (isMine (row, col+1, realBoard) == false)
                    playMinesweeperUtil(myBoard, realBoard, mines, row, col+1, m
            }

            //----------- 4th Neighbour (West) ------------

            // Only process this cell if this is a valid one
            if (isValid (row, col-1) == true)
            {
                   if (isMine (row, col-1, realBoard) == false)
                    playMinesweeperUtil(myBoard, realBoard, mines, row, col-1, m
            }

            //----------- 5th Neighbour (North-East) ------------

            // Only process this cell if this is a valid one
            if (isValid (row-1, col+1) == true)
            {
                if (isMine (row-1, col+1, realBoard) == false)
                    playMinesweeperUtil(myBoard, realBoard, mines, row-1, col+1,
            }

             //----------- 6th Neighbour (North-West) ------------

            // Only process this cell if this is a valid one
            if (isValid (row-1, col-1) == true)
            {
                 if (isMine (row-1, col-1, realBoard) == false)
                    playMinesweeperUtil(myBoard, realBoard, mines, row-1, col-1,
            }

             //----------- 7th Neighbour (South-East) ------------

            // Only process this cell if this is a valid one
            if (isValid (row+1, col+1) == true)
            {
                 if (isMine (row+1, col+1, realBoard) == false)
                    playMinesweeperUtil(myBoard, realBoard, mines, row+1, col+1,
            }

            //----------- 8th Neighbour (South-West) ------------

            // Only process this cell if this is a valid one
            if (isValid (row+1, col-1) == true)
            {
                if (isMine (row+1, col-1, realBoard) == false)
                    playMinesweeperUtil(myBoard, realBoard, mines, row+1, col-1,
            }
        }

        return (false);
    }
}

// A Function to place the mines randomly
// on the board
void placeMines(int mines[][2], char realBoard[][MAXSIDE])
{
    bool mark[MAXSIDE*MAXSIDE];

    memset (mark, false, sizeof (mark));
```

```c
    // Continue until all random mines have been created.
    for (int i=0; i<MINES; )
     {
        int random = rand() % (SIDE*SIDE);
        int x = random / SIDE;
        int y = random % SIDE;

        // Add the mine if no mine is placed at this
        // position on the board
        if (mark[random] == false)
        {
            // Row Index of the Mine
            mines[i][0]= x;
            // Column Index of the Mine
            mines[i][1] = y;

            // Place the mine
            realBoard[mines[i][0]][mines[i][1]] = '*';
            mark[random] = true;
            i++;
        }
    }

    return;
}

// A Function to initialise the game
void initialise(char realBoard[][MAXSIDE], char myBoard[][MAXSIDE])
{
    // Initiate the random number generator so that
    // the same configuration doesn't arises
    srand(time (NULL));

    // Assign all the cells as mine-free
    for (int i=0; i<SIDE; i++)
    {
        for (int j=0; j<SIDE; j++)
        {
            myBoard[i][j] = realBoard[i][j] = '-';
        }
    }

    return;
}

// A Function to cheat by revealing where the mines are
// placed.
void cheatMinesweeper (char realBoard[][MAXSIDE])
{
    printf ("The mines locations are-\n");
    printBoard (realBoard);
    return;
}

// A function to replace the mine from (row, col) and put
// it to a vacant space
void replaceMine (int row, int col, char board[][MAXSIDE])
{
    for (int i=0; i<SIDE; i++)
    {
        for (int j=0; j<SIDE; j++)
            {
                // Find the first location in the board
                // which is not having a mine and put a mine
                // there.
                if (board[i][j] != '*')
                {
                    board[i][j] = '*';
                    board[row][col] = '-';
                    return;
                }
            }
    }
    return;
}

// A Function to play Minesweeper game
void playMinesweeper ()
{
    // Initially the game is not over
    bool gameOver = false;

    // Actual Board and My Board
    char realBoard[MAXSIDE][MAXSIDE], myBoard[MAXSIDE][MAXSIDE];

    int movesLeft = SIDE * SIDE - MINES, x, y;
    int mines[MAXMINES][2]; // stores (x,y) coordinates of all mines.

      initialise (realBoard, myBoard);

    // Place the Mines randomly
    placeMines (mines, realBoard);

     /*
     If you want to cheat and know
     where mines are before playing the game
     then uncomment this part

     cheatMinesweeper(realBoard);
     */

    // You are in the game until you have not opened a mine
    // So keep playing

    int currentMoveIndex = 0;
    while (gameOver == false)
     {
        printf ("Current Status of Board : \n");
        printBoard (myBoard);
        makeMove (&x, &y);

        // This is to guarantee that the first move is
        // always safe
        // If it is the first move of the game
        if (currentMoveIndex == 0)
        {
            // If the first move itself is a mine
            // then we remove the mine from that location
            if (isMine (x, y, realBoard) == true)
                replaceMine (x, y, realBoard);
        }

        currentMoveIndex ++;

        gameOver = playMinesweeperUtil (myBoard, realBoard, mines, x, y, &movesL

        if ((gameOver == false) && (movesLeft == 0))
         {
            printf ("\nYou won !\n");
            gameOver = true;
         }
    }
    return;
}

// A Function to choose the difficulty level
// of the game
void chooseDifficultyLevel ()
{
    /*
    --> BEGINNER = 9 * 9 Cells and 10 Mines
    --> INTERMEDIATE = 16 * 16 Cells and 40 Mines
    --> ADVANCED = 24 * 24 Cells and 99 Mines
    */

    int level;

    printf ("Enter the Difficulty Level\n");
    printf ("Press 0 for BEGINNER (9 * 9 Cells and 10 Mines)\n");
    printf ("Press 1 for INTERMEDIATE (16 * 16 Cells and 40 Mines)\n");
    printf ("Press 2 for ADVANCED (24 * 24 Cells and 99 Mines)\n");

    scanf ("%d", &level);

    if (level == BEGINNER)
    {
        SIDE = 9;
        MINES = 10;
    }

    if (level == INTERMEDIATE)
    {
        SIDE = 16;
        MINES = 40;
    }

    if (level == ADVANCED)
    {
        SIDE = 24;
        MINES = 99;
    }

    return;
}

// Driver Program to test above functions
int main()
{
    /* Choose a level between
    --> BEGINNER = 9 * 9 Cells and 10 Mines
    --> INTERMEDIATE = 16 * 16 Cells and 40 Mines
    --> ADVANCED = 24 * 24 Cells and 99 Mines
    */
    chooseDifficultyLevel ();

    playMinesweeper ();

    return (0);
}
```

Run on IDE

Input:

```
0
1 2
2 3
3 4
4 5
```

Output:

```
Enter the Difficulty Level
```

```
Press 0 for BEGINNER (9 * 9 Cells and 10 Mines)
Press 1 for INTERMEDIATE (16 * 16 Cells and 40 Mines)
Press 2 for ADVANCED (24 * 24 Cells and 99 Mines)
Current Status of Board :
    0 1 2 3 4 5 6 7 8

0   - - - - - - - - -
1   - - - - - - - - -
2   - - - - - - - - -
3   - - - - - - - - -
4   - - - - - - - - -
5   - - - - - - - - -
6   - - - - - - - - -
7   - - - - - - - - -
8   - - - - - - - - -
Enter your move, (row, column) -> Current Status of Board :
    0 1 2 3 4 5 6 7 8

0   - - - - - - - - -
1   - - 2 - - - - - -
2   - - - - - - - - -
3   - - - - - - - - -
4   - - - - - - - - -
5   - - - - - - - - -
6   - - - - - - - - -
7   - - - - - - - - -
8   - - - - - - - - -
Enter your move, (row, column) ->     0 1 2 3 4 5 6 7 8

0   - - - - - - - * *
1   - - 2 * - - - - -
2   - - - * * - - - -
3   - - - - - - - * -
4   - - - - - - - - -
5   - - - - - - - - -
6   - - * - - - - - -
7   - - - - * - - * -
8   - * - - - - - - -

You lost!
```

C program implementation when user input is choose randomly

```
// A C++ Program to Implement and Play Minesweeper
// without taking input from user

#include<bits/stdc++.h>
using namespace std;

#define BEGINNER 0
#define INTERMEDIATE 1
#define ADVANCED 2
#define MAXSIDE 25
#define MAXMINES 99
#define MOVESIZE 526 // (25 * 25 - 99)

int SIDE ; // side length of the board
int MINES ; // number of mines on the board

// A Utility Function to check whether given cell (row, col)
// is a valid cell or not
bool isValid(int row, int col)
{
    // Returns true if row number and column number
    // is in range
    return (row >= 0) && (row < SIDE) &&
           (col >= 0) && (col < SIDE);
}

// A Utility Function to check whether given cell (row, col)
// has a mine or not.
bool isMine (int row, int col, char board[][MAXSIDE])
{
    if (board[row][col] == '*')
        return (true);
    else
        return (false);
}

// A Function to get the user's move and print it
// All the moves are assumed to be distinct and valid.
void makeMove (int *x, int *y, int moves[][2], int currentMoveIndex)
{
    *x = moves[currentMoveIndex][0];
    *y = moves[currentMoveIndex][1];

    printf ("\nMy move is (%d, %d)\n", *x, *y);

    /*
    // The above moves are pre-defined
    // If you want to make your own move
    // then uncomment this section and comment
    // the above section

      scanf("%d %d", x, y);
    */

    return;
}

// A Function to randomly assign moves
void assignMoves (int moves[][2], int movesLeft)
{
    bool mark[MAXSIDE*MAXSIDE];

    memset(mark, false, sizeof(mark));

    // Continue until all moves are assigned.
    for (int i=0; i<movesLeft; )
     {
        int random = rand() % (SIDE*SIDE);
        int x = random / SIDE;
        int y = random % SIDE;

        // Add the mine if no mine is placed at this
        // position on the board
        if (mark[random] == false)
        {
            // Row Index of the Mine
            moves[i][0]= x;
            // Column Index of the Mine
            moves[i][1] = y;

            mark[random] = true;
            i++;
        }
    }
    return;
}

// A Function to print the current gameplay board
void printBoard(char myBoard[][MAXSIDE])
{
    int i,j;

    printf ("    ");

    for (i=0; i<SIDE; i++)
        printf ("%d ", i);

    printf ("\n\n");

    for (i=0; i<SIDE; i++)
    {
        printf ("%d   ", i);

        for (j=0; j<SIDE; j++)
            printf ("%c ", myBoard[i][j]);
        printf ("\n");
    }
    return;
}

// A Function to count the number of
// mines in the adjacent cells
int countAdjacentMines(int row ,int col ,int mines[][2], char realBoard[][MAXSID
{

    int i;
    int count = 0;

    /*
        Count all the mines in the 8 adjacent
        cells

            N.W   N   N.E
              \   |   /
               \  |  /
            W----Cell----E
                 / | \
               /   |  \
            S.W    S   S.E

        Cell-->Current Cell (row, col)
        N -->  North        (row-1, col)
        S -->  South        (row+1, col)
        E -->  East         (row, col+1)
        W -->  West            (row, col-1)
        N.E--> North-East   (row-1, col+1)
        N.W--> North-West   (row-1, col-1)
        S.E--> South-East   (row+1, col+1)
        S.W--> South-West   (row+1, col-1)
    */

    //----------- 1st Neighbour (North) ------------

        // Only process this cell if this is a valid one
        if (isValid (row-1, col) == true)
        {
               if (isMine (row-1, col, realBoard) == true)
               count++;
        }

    //----------- 2nd Neighbour (South) ------------

        // Only process this cell if this is a valid one
        if (isValid (row+1, col) == true)
        {
               if (isMine (row+1, col, realBoard) == true)
               count++;
        }

    //----------- 3rd Neighbour (East) ------------

        // Only process this cell if this is a valid one
        if (isValid (row, col+1) == true)
        {
            if (isMine (row, col+1, realBoard) == true)
               count++;
        }

    //----------- 4th Neighbour (West) ------------

        // Only process this cell if this is a valid one
        if (isValid (row, col-1) == true)
        {
               if (isMine (row, col-1, realBoard) == true)
               count++;
        }

    //----------- 5th Neighbour (North-East) ------------

        // Only process this cell if this is a valid one
        if (isValid (row-1, col+1) == true)
        {
            if (isMine (row-1, col+1, realBoard) == true)
               count++;
        }

     //----------- 6th Neighbour (North-West) ------------

        // Only process this cell if this is a valid one
        if (isValid (row-1, col-1) == true)
        {
             if (isMine (row-1, col-1, realBoard) == true)
               count++;
        }

     //----------- 7th Neighbour (South-East) ------------

        // Only process this cell if this is a valid one
        if (isValid (row+1, col+1) == true)
        {
               if (isMine (row+1, col+1, realBoard) == true)
               count++;
        }

    //----------- 8th Neighbour (South-West) ------------

        // Only process this cell if this is a valid one
        if (isValid (row+1, col-1) == true)
        {
            if (isMine (row+1, col-1, realBoard) == true)
               count++;
        }

    return (count);
}

// A Recursive Fucntion to play the Minesweeper Game
bool playMinesweeperUtil(char myBoard[][MAXSIDE], char realBoard[][MAXSIDE],
            int mines[][2], int row, int col, int *movesLeft)
{

    // Base Case of Recursion
    if (myBoard[row][col]!='-')
```

```
        return (false);

    int i, j;

    // You opened a mine
    // You are going to lose
    if (realBoard[row][col] == '*')
    {
        myBoard[row][col]='*';

        for (i=0; i<MINES; i++)
            myBoard[mines[i][0]][mines[i][1]]='*';

        printBoard (myBoard);
        printf ("\nYou lost!\n");
        return (true) ;
    }

    else
    {

        // Calculate the number of adjacent mines and put it
        // on the board.
        int count = countAdjacentMines(row, col, mines, realBoard);
        (*movesLeft)--;

        myBoard[row][col] = count + '0';

        if (!count)
        {
            /*
            Recur for all 8 adjacent cells

                N.W   N   N.E
                  \   |   /
                      \  |  /
                W----Cell----E
                     / | \
                   /   |  \
                S.W    S   S.E

            Cell-->Current Cell (row, col)
            N -->  North        (row-1, col)
            S -->  South        (row+1, col)
            E -->  East         (row, col+1)
            W -->  West            (row, col-1)
            N.E--> North-East   (row-1, col+1)
            N.W--> North-West   (row-1, col-1)
            S.E--> South-East   (row+1, col+1)
            S.W--> South-West   (row+1, col-1)
            */

                //----------- 1st Neighbour (North) ------------

            // Only process this cell if this is a valid one
            if (isValid (row-1, col) == true)
            {
                   if (isMine (row-1, col, realBoard) == false)
                   playMinesweeperUtil(myBoard, realBoard, mines, row-1, col, mo
            }

            //----------- 2nd Neighbour (South) ------------

            // Only process this cell if this is a valid one
            if (isValid (row+1, col) == true)
            {
                   if (isMine (row+1, col, realBoard) == false)
                    playMinesweeperUtil(myBoard, realBoard, mines, row+1, col, m
            }
            //----------- 3rd Neighbour (East) ------------

            // Only process this cell if this is a valid one
            if (isValid (row, col+1) == true)
            {
                if (isMine (row, col+1, realBoard) == false)
                    playMinesweeperUtil(myBoard, realBoard, mines, row, col+1, m
            }

            //----------- 4th Neighbour (West) ------------

            // Only process this cell if this is a valid one
            if (isValid (row, col-1) == true)
            {
                   if (isMine (row, col-1, realBoard) == false)
                    playMinesweeperUtil(myBoard, realBoard, mines, row, col-1, m
            }

            //----------- 5th Neighbour (North-East) ------------

            // Only process this cell if this is a valid one
            if (isValid (row-1, col+1) == true)
            {
                if (isMine (row-1, col+1, realBoard) == false)
                    playMinesweeperUtil(myBoard, realBoard, mines, row-1, col+1,
            }

             //----------- 6th Neighbour (North-West) ------------

            // Only process this cell if this is a valid one
            if (isValid (row-1, col-1) == true)
            {
                 if (isMine (row-1, col-1, realBoard) == false)
                    playMinesweeperUtil(myBoard, realBoard, mines, row-1, col-1,
            }

             //----------- 7th Neighbour (South-East) ------------

            // Only process this cell if this is a valid one
            if (isValid (row+1, col+1) == true)
            {
                   if (isMine (row+1, col+1, realBoard) == false)
                    playMinesweeperUtil(myBoard, realBoard, mines, row+1, col+1,
            }

            //----------- 8th Neighbour (South-West) ------------

            // Only process this cell if this is a valid one
            if (isValid (row+1, col-1) == true)
            {
                if (isMine (row+1, col-1, realBoard) == false)
                    playMinesweeperUtil(myBoard, realBoard, mines, row+1, col-1,
            }
        }

        return (false);
    }
}

// A Function to place the mines randomly
// on the board
void placeMines(int mines[][2], char realBoard[][MAXSIDE])
{
    bool mark[MAXSIDE*MAXSIDE];

    memset (mark, false, sizeof (mark));

    // Continue until all random mines have been created.
    for (int i=0; i<MINES; )
     {
```

```c
        int random = rand() % (SIDE*SIDE);
        int x = random / SIDE;
        int y = random % SIDE;

        // Add the mine if no mine is placed at this
        // position on the board
        if (mark[random] == false)
        {
            // Row Index of the Mine
            mines[i][0]= x;
            // Column Index of the Mine
            mines[i][1] = y;

            // Place the mine
            realBoard[mines[i][0]][mines[i][1]] = '*';
            mark[random] = true;
            i++;
        }
    }

    return;
}

// A Function to initialise the game
void initialise (char realBoard[][MAXSIDE], char myBoard[][MAXSIDE])
{
    // Initiate the random number generator so that
    // the same configuration doesn't arises
    srand (time (NULL));

    // Assign all the cells as mine-free
    for (int i=0; i<SIDE; i++)
    {
        for (int j=0; j<SIDE; j++)
        {
            myBoard[i][j] = realBoard[i][j] = '-';
        }
    }

    return;
}

// A Function to cheat by revealing where the mines are
// placed.
void cheatMinesweeper (char realBoard[][MAXSIDE])
{
    printf ("The mines locations are-\n");
    printBoard (realBoard);
    return;
}

// A function to replace the mine from (row, col) and put
// it to a vacant space
void replaceMine (int row, int col, char board[][MAXSIDE])
{
    for (int i=0; i<SIDE; i++)
    {
        for (int j=0; j<SIDE; j++)
            {
                // Find the first location in the board
                // which is not having a mine and put a mine
                // there.
                if (board[i][j] != '*')
                {
                    board[i][j] = '*';
                    board[row][col] = '-';
                    return;
                }
            }
    }
    return;
}

// A Function to play Minesweeper game
void playMinesweeper ()
{
    // Initially the game is not over
    bool gameOver = false;

    // Actual Board and My Board
    char realBoard[MAXSIDE][MAXSIDE], myBoard[MAXSIDE][MAXSIDE];

    int movesLeft = SIDE * SIDE - MINES, x, y;
    int mines[MAXMINES][2]; // Stores (x, y) coordinates of all mines.
      int moves[MOVESIZE][2]; // Stores (x, y) coordinates of the moves

      // Initialise the Game
      initialise (realBoard, myBoard);

    // Place the Mines randomly
    placeMines (mines, realBoard);

    // Assign Moves
    // If you want to make your own input move,
    // then the below function should be commnented
    assignMoves (moves, movesLeft);

    /*
     //If you want to cheat and know
     //where mines are before playing the game
     //then uncomment this part

     cheatMinesweeper(realBoard);
    */

    // You are in the game until you have not opened a mine
    // So keep playing

    int currentMoveIndex = 0;
    while (gameOver == false)
     {
        printf ("Current Status of Board : \n");
        printBoard (myBoard);

        makeMove (&x, &y, moves, currentMoveIndex);

        // This is to guarantee that the first move is
        // always safe
        // If it is the first move of the game
        if (currentMoveIndex == 0)
        {
            // If the first move itself is a mine
            // then we remove the mine from that location
            if (isMine (x, y, realBoard) == true)
                replaceMine (x, y, realBoard);
        }

        currentMoveIndex ++;

        gameOver = playMinesweeperUtil (myBoard, realBoard, mines, x, y, &movesL

        if ((gameOver == false) && (movesLeft == 0))
         {
            printf ("\nYou won !\n");
            gameOver = true;
         }
    }

    return;
}
```

```
// A Function to choose the difficulty level
// of the game
void chooseDifficultyLevel (int level)
{
    /*
    --> BEGINNER = 9 * 9 Cells and 10 Mines
    --> INTERMEDIATE = 16 * 16 Cells and 40 Mines
    --> ADVANCED = 24 * 24 Cells and 99 Mines
    */

    if (level == BEGINNER)
    {
        SIDE = 9;
        MINES = 10;
    }

    if (level == INTERMEDIATE)
    {
        SIDE = 16;
        MINES = 40;
    }

    if (level == ADVANCED)
    {
        SIDE = 24;
        MINES = 99;
    }

    return;
}

// Driver Program to test above functions
int main()
{
    /* Choose a level between
    --> BEGINNER = 9 * 9 Cells and 10 Mines
    --> INTERMEDIATE = 16 * 16 Cells and 40 Mines
    --> ADVANCED = 24 * 24 Cells and 99 Mines
    */
    chooseDifficultyLevel (BEGINNER);

    playMinesweeper ();
    return (0);
}
```

Run on IDE

Output:

```
Current Status of Board :
    0 1 2 3 4 5 6 7 8


0   - - - - - - - - -
1   - - - - - - - - -
2   - - - - - - - - -
3   - - - - - - - - -
4   - - - - - - - - -
5   - - - - - - - - -
6   - - - - - - - - -
7   - - - - - - - - -
8   - - - - - - - - -
```

```
My move is (4, 7)
Current Status of Board :
    0 1 2 3 4 5 6 7 8

0   - - - - - - - - -
1   - - - - - - - - -
2   - - - - - - - - -
3   - - - - - - - - -
4   - - - - - - - 2 -
5   - - - - - - - - -
6   - - - - - - - - -
7   - - - - - - - - -
8   - - - - - - - - -

My move is (3, 7)
Current Status of Board :
    0 1 2 3 4 5 6 7 8

0   - - - - - - - - -
1   - - - - - - - - -
2   - - - - - - - - -
3   - - - - - - - 1 -
4   - - - - - - - 2 -
5   - - - - - - - - -
6   - - - - - - - - -
7   - - - - - - - - -
8   - - - - - - - - -

My move is (7, 3)
    0 1 2 3 4 5 6 7 8

0   - - - - - - - - -
1   - - - - * - - - -
2   - - - - * - - - -
3   - - - - - - - 1 -
4   - - - * - - * 2 -
5   - - - - - - - * -
6   * - * - - - * - -
7   - - - * * - - - -
8   - - - - - - - - -

You lost!
```

This article is contributed by **Rachit Belwariar**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

1 Comment Category: Project

Related Posts:

- Creating a C/C++ Code Formatting tool with help of Clang tools
- OpenCV C++ Program for coin detection
- Basic Graphic Programming in C++
- Cartooning an Image using OpenCV – Python
- Creating a PortScanner in C
- Creating a Proxy Webserver in Python | Set 2
- OpenCV C++ Program to blur a Video
- A Number Link Game

(Login to Rate and Mark)

3.4 Average Difficulty : **3.4/5.0**
Based on **5** vote(s)

🟩 Add to TODO List
🟩 Mark as DONE

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

1 Comment **GeeksforGeeks** 1 **Login**

Recommend **Share** **Sort by Newest**

Join the discussion…

Atinesh Si · 4 days ago
That's a big code
Reply · Share ›