Developer’s Documentation

Name: Emil Badraddinli

Neptun Code: I9LZOM

Project: Minesweeper

• Algorithm of the program

- Board’s size and number of mines is determined based on user’s input parameters. Program asks user to enter coordinates of the cell which is suspected to user. Then program will examine the element of array with same coordinates which is given by the user. In case of cell contains mine, the program will terminate and there will be losing message on the screen, as well as, the amount of time user spent on the playing game. Otherwise, it will show the number of mines in the adjacency cells. If the number is 0, program will reveal all mine-free cells automatically. Every time program checks if all mine-free cells are uncovered or not. In case of yes, user wins the game and there will be winning message on the screen and amount of time user spent on playing the game.

• Solution of the algorithm

There are 3 modules in the program. 2 source files and 1 header file.

- **Main.c, MineSweeper.c, MineSweeper.h.**

Main.c: Consists of only one main function.

MineSweeper. C: Contains function definitions and two global variables.

- **Global variables**:

```c
bool lose = false;
bool win = false;
```

MineSweeper.h: Contains system header files, structures and functions declarations.

- **Header files:**

  #include <stdbool.h>

#include <stdlib.h>

#include <stdio.h>

#include <time.h>

• **The structures** which is used in the program:

```
  typedef struct Cell{
    int number_of_mines;
    char ch;
    bool uncovered;
    bool bomb;
}Cell;
```

First one is the **Cell** which contains information about the cell:

- Number of bombs which are in the adjacent cells
- Character of the cell (it can change during the game as **'-'**, **'+' '!'** or **number**)
- Boolean type for the uncovered (Cell can be covered or uncovered)
- Boolean type for the bomb (Cell can contain mine or not)

```
typedef struct Board{
    int height, width;
    Cell **cells;
}Board;
```

Second one is the **Board.** Structure members are the following:

- Parameters for the size of the board (height and weight) which is determined by the user before the starting game
- Pointer to the pointer in order to create 2 dimensional array of cells for board

• **Global Variables:**

- There are 2 boolean type global variables called **win** and **lose**. Initially they are assigned to the false because there is no information about the user's status whether he or she win or lose before the game.

- **Function prototypes** (Declarations)

```
void make_board(Board*);
```

Parameters: Pointer to the board in order to create board.

```
void print_board(Board*);
```

Parameters: Pointer to the board in order to print board

```
void bombplacing_randomly(Board*, int);
```

Parameters: pointer to the board and number of bombs which will be in the board

```
void numof_adjacent_mines(Board*);
```

Parameters: pointer to the board. And determine number of bombs in neighbors of the cell.

```
void uncover(Board*, int, int);
```

Parameters: pointer to the board, selected coordinates by the user for uncovering cells

```
void reveal_automatically(Board*, int, int);
```

Parameters: pointer to the board and cell's coordinates

```
void check_for_win(Board*, int);
```

Parameters: pointer to the board and number of mines

```
void check_for_win(Board*, int);
```

Parameters: Receiving no parameter. It is a function to play the game

- Function definitions (How do they works in the program?)

- `void make_board(Board *ptr)`

  - initially, it is created array of pointers using dynamic memory. Then in the **for** loop it is made cells in the board.

```
    ptr->cells=(Cell**)malloc((ptr->height+2)*sizeof(Cell*));
   for(i = 0; i <= ptr->height+1; ++i)
     ptr->cells[i]=(Cell*)malloc((ptr->width+2)*sizeof(Cell));
```

In the next process, using nested loop, assigning initial values to the struct elements according to their data types. Giving initialization is based on game logic.

- `void print_board(Board *ptr)`

  - The aim of this function is to print the board and put the numbers for columns and rows which indicates coordinates of cell on the screen. Firstly, using single line **for** loop to put numbers in the each row

```
   for(i = 1; i <= ptr->width; ++i)
       printf("%d ", i);
```

Then with the help of the nested loop, to print numbers for columns and characters for cells in the board. Initially, character in the board is **'-'** .

```
for(i = 1; i <= ptr->height; ++i){
       for(j = 0; j<= ptr->width; ++j){
         if(j == 0) printf("%d ", i);
         else printf("%c ", ptr->cells[i][j].ch);
       }
```

- `void bombplacing_randomly(Board *ptr, int mines)`

  - Before starting game, program puts mines in the board in random order. The important thing is to use built in function which is called **rand().** The random number is generated as shown below

```
-   random_row = rand()%ptr->height+1;
-   random_col = rand()%ptr->width+1;
```

  Rand() function produces random number between 0 ≤ x ≤ height and 0 ≤ y ≤ width. The meaning of adding one to height and width is that function works with mod. It produces number from 0 up to height and width. That is why it is added 1 to the size parameters.

  After generating random numbers program checks if produced coordinates contain mine or not. If not make it a bomb and increment number of mines

```
if(ptr->cells[random_row][random_col].bomb == false && (random_row != 0 && random_col != 0))
        {
            ptr->cells[random_row][random_col].bomb = true;
            num_of_mine++;
        }
```

The incrementing process terminates when the **num_of_mine** is equal to **mines** which is determined by the user.

- `void numof_adjacent_mines(Board *ptr)`
    - It is used to calculate number of mines in adjacent cell. In order to create function it is better to use nested loops. If there is a bomb increment the mines.

```
for(m = i-1; m <= i+1; ++m)
 for(n = j-1; n <= j+1; ++n)
    if(ptr->cells[m][n].bomb == true)
      ptr->cells[i][j].number_of_mines++;
```

- `void uncover(Board *ptr, int a, int b)`
    - Function checks if the selected coordinate contains a bomb, player will lose the game. Boolean variable **lose** becomes true. Nested loop to uncover the cells. If there is a bomb, show the bomb sign **'*'**, otherwise function will show the number of bombs in the surrounding 8 cells.

```
for(i = 1; i <= ptr->height; ++i)
 for(j = 1; j <= ptr->width; ++j)
   if(ptr->cells[i][j].bomb == true)
      ptr->cells[i][j].ch = '*';
   else
      ptr->cells[i][j].ch = ptr->cells[i][j].number_of_mines + '0';
```

    - If the number is zero, means there is no bomb in adjacent cells. Function will execute another function called **reveal_automatically** (explanation in the next page)
    - Then function calls **print_board** function in order to show the board on the screen. User also gets message **"You lost"** on the screen.

- `void reveal_automatically(Board *ptr, int a, int b)`
    - If selected coordinates are not uncovered function makes it covered. Using nested loop function uncovers adjacent cells which is uncovered . In order to do that function calls previous function called **uncover**.

```
if(ptr->cells[a][b].uncovered == false)
   {
       ptr->cells[a][b].uncovered = true;
       for(i = a-1; i <= a+1; ++i)
           for(j = b-1; j <= b+1; ++j)
              if(ptr->cells[i][j].uncovered == false)
                uncover(ptr,i,j);
   }
```

- `void check_for_win(Board *ptr, int mines)`

  - At first, function assign zero to **counter** variable. Counter variable holds number of cells which does not contain bomb. Using nested loop, if there is no bomb then increment the **counter**

```
for(i = 1; i <= ptr->height; ++i)
 for(j = 1; j <= ptr->width; ++j)
   if(ptr->cells[i][j].bomb==false && ptr->cells[i][j].ch != '-' && ptr-
>cells[i][j].ch != '!')
     counter++;
```

  - If the **counter** is equal to this equation: (ptr->height*ptr->width) – mines. It means that ptr->height * ptr->width is the maximum number of bombs that can be placed in the board. Mines variable holds number of mines which is entered by the user. If above equation satisfies then player found all the cells which does not contain bomb. Boolean variable **win** becomes true.

  - Function will also show the bomb sign, number of mines in the surrounding cells as well as the winning message like **“You won”.** Function will call **print_board** function

```
if (counter == (ptr->height*ptr->width) - mines)
    {
        win = true;
        for(i = 1; i <= ptr->height; ++i)
            for(j = 1; j <= ptr->width; ++j)
               if(ptr->cells[i][j].bomb == true)
                  ptr->cells[i][j].ch = '*';
               else
                  ptr->cells[i][j].ch = ptr->cells[i][j].number_of_mines + '0';

        print_board(ptr);
        printf("\nYou Won, Congratulations!!!\n");
    }
```

- `void play_game()`

  - This function is the lifeline of the program. All the above functions are called here. First, setting seed for **rand()** which is used in the **bombplacing_randomly** function.

```
        srand(time(NULL));
```

Then it is asked to enter the size of the board(height and width respectively) and number of mines which will be placed in the board. After that it is created dynamic array. Now **make_board**, **bombplacing_randomly** and **numof_adjacent_mines** functions are called respectively.

In the **do while loop**, firstly, **print_board** function is called, then program asks user to mark(**m**) or uncover(**u**) the cell, after choosing operation user must enter the coordinates of the cell he or she want to uncover or mark. If the operation is uncovering the cell then **uncover** function is called, in case of marking, program puts **'!'** mark on the cell. At the end **check_for_win** function is called. If the all the mine-free cells are uncovered, user wins the game. The memory is released.

Finally, the solution of algorithm finishes in the main function by calling **play_game** function.

One more thing is to calculate how much time user spent on the game. In order to do that the program gets system time at the beginning and at the end of the game. Then determine the difference of these time intervals and print it on the screen.

```
int main()
{
    clock_t begin = clock(); /* getting time at the beginning */

    play_game();

    clock_t end = clock();  /* getting time at the end */
    double spend_time = (double)(end - begin) / CLOCKS_PER_SEC; /* time difference */

    printf("You spent %.2f seconds playing the game\n", spend_time);
    return 0;
}
```