

```
#include <iostream>
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <ctype.h>
```

```
#include <time.h>
```

```
using namespace std;
```

```
#define BOARD_SIZE_b 6
```

```
#define BOARD_SIZE_a 10
```

```
#define BOARD_SIZE_p 15
```

```
#define TRUE 0
```

```
#define FALSE 1
```

```
// I refuse to have to read in the game board for every function that calls it..
```

```
// Global "lost" variable is kind of stupid, but, it works...
```

```
char board_b[BOARD_SIZE_b][BOARD_SIZE_b];
```

```
char game_board_b[BOARD_SIZE_b][BOARD_SIZE_b];
```

```
char board_a[BOARD_SIZE_a][BOARD_SIZE_a];
```

```
char game_board_a[BOARD_SIZE_a][BOARD_SIZE_a];
```

```
char board_p[BOARD_SIZE_p][BOARD_SIZE_p];
```

```
char game_board_p[BOARD_SIZE_p][BOARD_SIZE_p];
```

```
int lost = 0;
```

```
/* Function prototypes */
```

```
void display_welcome();
```

```
void build_board_b();
```

```
void build_board_a();
```

```
void build_board_p();
```

```
void build_gboard_b();
```

```
void build_gboard_a();
```

```
void build_gboard_p();
```

```
void create_mines_b();
```

```
void create_mines_a();
```

```
void create_mines_p();
```

```
void print_board_b();
```

```
void print_board_a();
```

```
void print_board_p();
```

```
void print_fullboard_b();
```

```
void print_fullboard_a();
```

```
void print_fullboard_p();
```

```
void start_b();
```

```
void start_a();
```

```
void start_p();
```

```
int play_game_b();
```

```
int play_game_a();
```

```
int play_game_p();
```

```
void play_again_b();
```

```
void play_again_a();
```

```
void play_again_p();
```

```
int check_win_game_b();
```

```
int check_win_game_a();
```

```
int check_win_game_p();
```

```
void check_for_mine_b(int, int);
```

```
void check_for_mine_a(int, int);
```

```
void check_for_mine_p(int, int);
```

```
int check_for_nearby_mines_b(int, int);
```

```
int check_for_nearby_mines_a(int, int);
```

```
int check_for_nearby_mines_p(int, int);
```

```

// Main function

int main ()

{

    display_welcome();

    printf("\n If you are ready to play Minesweeper.....    Just press ENTER. :)");

    getchar();

    system("cls");//clear screen

    printf("\n\n Please Choose Your Level you want to play from the list below. \n ( Please Enter The No.
Preeceding Your Choice ) \n\n");

    printf(" 1. Beginner Level \n\n 2. Amateur Level \n\n 3. Professional Level \n\n 4. Exit Minesweeper
\n\n");

    int choice;

    scanf ("%d",&choice);

    switch (choice)
    {

    case 1:

        system("cls");

        cout<<"Beginner Level Starting...";

        start_b();

        break;

    case 2:

        system("cls");

        cout<<"Amateur Level Starting...";

        start_p();

```

```
break;
```

case 3:

```
system("cls");
```

```
cout<<"Professional Level Starting...";
```

```
start_p();
```

```
break;
```

default:

```
system("cls");
```

```
cout<<"Exiting Minesweeper... \n";
```

```
return 0;
```

```
}
```

```
return 0;
```

```
}
```

```
/* Build board used for created random mines */
```

```
void build_board_b()
```

```
{
```

```
int i, j; // Assign char for board elements
```

```
for(i = 0; i < BOARD_SIZE_b; i++)
```

```
    for(j = 0; j < BOARD_SIZE_b; j++)
```

```
        board_b[i][j] = 'o';
```

```
// Place mines in this board, it remains hidden from user until the game has finished.
```

```
    create_mines_b();  
}
```

```
void build_board_a()  
{  
    int i, j; // Assign char for board elements  
    for(i = 0; i < BOARD_SIZE_a; i++)  
        for(j = 0; j < BOARD_SIZE_a; j++)  
            board_a[i][j] = 'o';  
    // Place mines in this board, it remains hidden from user until the game has finished.  
    create_mines_a();  
}
```

```
void build_board_p()  
{  
    int i, j; // Assign char for board elements  
    for(i = 0; i < BOARD_SIZE_p; i++)  
        for(j = 0; j < BOARD_SIZE_p; j++)  
            board_p[i][j] = 'o';  
    // Place mines in this board, it remains hidden from user until the game has finished.  
    create_mines_p();  
}
```

```
/* Build game board for user input */
```

```
void build_gboard_b()
```

```

{

    int i, j;

    int row, col;

    cout<<" Creating Beginner Game Board... \n Ready... \n Set... \n PLAY MINESWEEPER !!!! \n\n";

    // Assign char 'o' for all board elements

    for(i = 0; i < BOARD_SIZE_b; i++)

        for(j = 0; j < BOARD_SIZE_b; j++)

            game_board_b[i][j] = 'o';

    // Print board

    for(col = 0; col < BOARD_SIZE_b; col++)

        cout<<(col + 1)<<" ";

    cout<<"\n\n";

    for(row = 0; row < BOARD_SIZE_b; row++)

    {

        for(col = 0; col < BOARD_SIZE_b; col++)

        {

            cout<<game_board_b[row][col]<<" ";

        }

        cout<<" "<<(row + 1);

        cout<<"\n";

    }
}

```

```
}
```

```
void build_gboard_a()
```

```
{
```

```
    int i, j;
```

```
    int row, col;
```

```
    printf(" Creating Beginner Game Board... \n Ready... \n Set... \n PLAY MINESWEEPER !!!! \n\n");
```

```
    // Assign char 'o' for all board elements
```

```
    for(i = 0; i < BOARD_SIZE_a; i++)
```

```
        for(j = 0; j < BOARD_SIZE_a; j++)
```

```
            game_board_a[i][j] = 'o';
```

```
    // Print board
```

```
    for(col = 0; col < BOARD_SIZE_a; col++)
```

```
        printf("%d ", col + 1);
```

```
    printf("\n\n");
```

```
    for(row = 0; row < BOARD_SIZE_a; row++)
```

```
    {
```

```
        for(col = 0; col < BOARD_SIZE_a; col++)
```

```
        {
```

```
            printf("%c ", game_board_a[row][col]);
```

```
        }
```

```
        printf(" %d ", row + 1);

        printf("\n");
    }
}
```

```
void build_gboard_p()
```

```
{
    int i, j;

    int row, col;

    printf(" Creating Beginner Game Board... \n Ready... \n Set... \n PLAY MINESWEEPER !!!! \n\n");

    // Assign char 'o' for all board elements
    for(i = 0; i < BOARD_SIZE_p; i++)
        for(j = 0; j < BOARD_SIZE_p; j++)
            game_board_p[i][j] = 'o';

    // Print board
    for(col = 0; col < BOARD_SIZE_p; col++)
        printf("%d ", col + 1);

    printf("\n\n");

    for(row = 0; row < BOARD_SIZE_p; row++)
    {
        for(col = 0; col < BOARD_SIZE_p; col++)
```

```
{  
    printf("%c ", game_board_p[row][col]);  
}  
printf(" %d ", row + 1);  
printf("\n");  
}  
}
```

/* Create random places in the array for mines */

void create_mines_b()

```
{  
    int i, random;  
  
    // Seeding srand() with time(0) so that mine locations aren't the same every time the game is played.  
    srand(time(0));  
  
    for (i = 0; i < BOARD_SIZE_b; i++)  
    {  
        random = rand() % (BOARD_SIZE_b);  
        board_b[random][i] = '*';  
    }  
}
```

void create_mines_a()

```
{  
    int i, random;
```

```

// Seeding srand() with time(0) so that mine locations aren't the same every time the game is played.
srand(time(0));

for (i = 0; i < BOARD_SIZE_a; i++)
{
    random = rand() % (BOARD_SIZE_a);
    board_a[random][i] = '*';
}
}

void create_mines_p()
{
    int i, random;

    // Seeding srand() with time(0) so that mine locations aren't the same every time the game is played.
    srand(time(0));

    for (i = 0; i < BOARD_SIZE_p; i++)
    {
        random = rand() % (BOARD_SIZE_p);
        board_p[random][i] = '*';
    }
}

/* Print the game board */
void print_board_b()

```

```

{
    int row, col;

    system("cls");

    for(col = 0; col < BOARD_SIZE_b; col++)

        printf("%d ", col + 1);

    printf("\n\n");

    for(row = 0; row < BOARD_SIZE_b; row++)
    {
        for(col = 0; col < BOARD_SIZE_b; col++)

            {
                printf("%c ", game_board_b[row][col]);

            }

        printf(" %d ", row + 1);

        printf("\n");
    }
}

```

```

void print_board_a()

{
    int row, col;

    system("cls");

    for(col = 0; col < BOARD_SIZE_a; row++)

```

```
printf("%d ", col + 1);
```

```
printf("\n\n");
```

```
for(row = 0; row < BOARD_SIZE_a; row++)
```

```
{
```

```
for(col = 0; col < BOARD_SIZE_a; col++)
```

```
{
```

```
printf("%c ", game_board_a[row][col]);
```

```
}
```

```
printf(" %d ", row + 1);
```

```
printf("\n");
```

```
}
```

```
}
```

```
void print_board_p()
```

```
{
```

```
int row, col;
```

```
system("cls");
```

```
for(col = 0; col < BOARD_SIZE_p; col++)
```

```
printf("%d ", col + 1);
```

```
printf("\n\n");
```

```
for(row = 0; row < BOARD_SIZE_p; row++)
```

```
{
```

```
    for(col = 0; col < BOARD_SIZE_p; col++)
    {
        printf("%c ", game_board_p[row][col]);
    }

    printf(" %d ", row + 1);

    printf("\n");
}
}
```

/* Print the full board showing mines */

```
void print_fullboard_b()
{
    int row, col;

    system("cls");

    for(col = 0; col < BOARD_SIZE_b; col++)

        printf("%d ", col + 1);

    printf("\n\n");

    for(row = 0; row < BOARD_SIZE_b; row++)
    {
        for(col = 0; col < BOARD_SIZE_b; col++)
        {
            printf("%c ", board_b[row][col]);
        }
    }
}
```

```
        printf(" %d ", row + 1);  
        printf("\n");  
    }  
}
```

```
void print_fullboard_a()  
{  
    int row, col;  
  
    system("cls");  
    for(col = 0; col < BOARD_SIZE_a; col++)  
        printf("%d ", col + 1);  
  
    printf("\n\n");  
    for(row = 0; row < BOARD_SIZE_a; row++)  
    {  
        for(col = 0; col < BOARD_SIZE_a; col++)  
        {  
            printf("%c ", board_a[row][col]);  
        }  
        printf(" %d ", row + 1);  
        printf("\n");  
    }  
}
```

```

void print_fullboard_p()
{
    int row, col;

    system("cls");

    for(col = 0; col < BOARD_SIZE_p; col++)

        printf("%d ", col + 1);

    printf("\n\n");

    for(row = 0; row < BOARD_SIZE_p; row++)
    {
        for(col = 0; col < BOARD_SIZE_p; col++)

            {

                printf("%c ", board_p[row][col]);

            }

        printf(" %d ", row + 1);

        printf("\n");

    }
}

```

/* Take user input for playing of the game */

```

int play_game_b()
{
    int r_selection = 0, c_selection = 0,

        nearbymines = 0, nearbymines2 = 0,

```

```

    nearbymines3 = 0, nearbymines4 = 0,

    nearbymines5 = 0, nearbymines6 = 0,

    nearbymines7 = 0, nearbymines8 = 0,

    i = 0;

// Recieves data from the user, first the row number, then the column number...

// I could think of other ways to do it, but this one seemed easiest.


do {

printf("\nMake a selection (ie. row [ENTER] col): \n");

printf("Row--> ");

scanf("%d", &r_selection);

printf("Col--> ");

scanf("%d", &c_selection);

} while(r_selection < 1 || r_selection > BOARD_SIZE_b || c_selection < 1 || c_selection >
BOARD_SIZE_b);

// ^ Checks for any invalid input statements from user.


check_for_mine_b(r_selection - 1, c_selection - 1);


if(lost == 1)

    return -1;

// Checks for nearby mines at every direction from user input location.

//Assigns that location the number of mines found nearby, updating the board.


nearbymines = check_for_nearby_mines_b(r_selection - 1, c_selection - 1);

```

```

game_board_b[r_selection - 1][c_selection - 1] = (char)( ((int)'0') + nearbymines );

// The following checks for mines nearby elements
// in the array with no mines.. it's a continuous
// loop until either a mine is found, or we
// reach the end of the array & it cannot be checked any further.

// It also changes the game_board[] with '0' if no mines are found. Very useful piece of code.
// It checks all elements left, right, up, down and all diagonal directions.
// By running a function that checks these same directions.
// A bit much to follow, though. I'm sure there's a much better way. I just don't know it yet.

if(nearbymines == 0)
{
    if(c_selection != BOARD_SIZE_b)
    {
        i = 0;
        while(nearbymines == 0 && (c_selection - 1 + i) < BOARD_SIZE_b)
        {
            // This is checking elements to the right
            nearbymines = check_for_nearby_mines_b(r_selection - 1, (c_selection - 1 + i));
            if(nearbymines != -1)
            {
                game_board_b[r_selection - 1][(c_selection - 1) + i] = (char) ( ((int)'0') + nearbymines );
                i++;
            }
        }
    }
}

```

```

    }
}

if(r_selection != 1)
{
    i = 0;

    while(nearbymines5 == 0 && (c_selection - 1 + i) < BOARD_SIZE_b && (r_selection - 1 - i) > 0)
    {
        // This is checking elements to the diagonal-uright
        nearbymines5 = check_for_nearby_mines_b((r_selection - 1 - i), (c_selection - 1 + i));

        if(nearbymines5 != -1)
        {
            game_board_b[(r_selection - 1) - i][(c_selection - 1) + i] = (char) ( ((int)'0') + nearbymines5);

            i++;
        }
    }
}

if(r_selection != BOARD_SIZE_b)
{
    i = 0;

    while(nearbymines6 == 0 && (r_selection - 1 + i) < BOARD_SIZE_b && (c_selection - 1 + i) <
BOARD_SIZE_b )
    {
        // This is checking elements to the diagonal-drighr
        nearbymines6 = check_for_nearby_mines_b((r_selection - 1 + i), (c_selection - 1 + i));

        if(nearbymines6 != -1)
        {

```

```
        game_board_b[(r_selection - 1) + i][(c_selection - 1) + i] = (char) ( ((int)'0') +  
nearbymines6);
```

```
        i++;
```

```
    }
```

```
}
```

```
}
```

```
}
```

```
if(r_selection != BOARD_SIZE_b)
```

```
{
```

```
    i = 0;
```

```
    while(nearbymines2 == 0 && (r_selection - 1 + i) < BOARD_SIZE_b)
```

```
    {
```

```
        // This is checking elements heading down
```

```
        nearbymines2 = check_for_nearby_mines_b((r_selection - 1 + i), c_selection - 1);
```

```
        if(nearbymines2 != -1)
```

```
        {
```

```
            game_board_b[(r_selection - 1) + i][c_selection - 1] = (char) ( ((int)'0') + nearbymines2 );
```

```
            i++;
```

```
        }
```

```
    }
```

```
if(c_selection != BOARD_SIZE_b)
```

```
{
```

```
    i = 0;
```

```
    while(nearbymines7 == 0 && (r_selection - 1 + i) < BOARD_SIZE_b && (c_selection - 1 - i) > 0)
```

```

{
    // This is checking elements to the diagonal-dleft
    nearbymines7 = check_for_nearby_mines_b((r_selection - 1 + i), (c_selection - 1 - i));
    if(nearbymines7 != -1)
    {
        game_board_b[(r_selection - 1) + i][(c_selection - 1) - i] = (char) ( ((int)'0') + nearbymines7);
        i++;
    }
}
}
}

```

```

if(r_selection != 1)
{
    i = 0;
    while(nearbymines3 == 0 && (r_selection - i) > 0)
    {
        // This is checking elements heading up
        nearbymines3 = check_for_nearby_mines_b((r_selection - 1 - i), c_selection - 1);
        if(nearbymines3 != -1)
        {
            game_board_b[(r_selection - 1) - i][c_selection - 1] = (char) ( ((int)'0') + nearbymines3 );
            i++;
        }
    }
}

```

```

if(c_selection != BOARD_SIZE_b)
{
    while(nearbymines8 == 0 && (c_selection - 1 - i) > 0 && (r_selection - 1 - i) > 0)
    {
        // This is checking elements to the diagonal-uleft
        nearbymines8 = check_for_nearby_mines_b((r_selection - 1 - i), (c_selection - 1 - i));
        if(nearbymines8 != -1)
        {
            game_board_b[(r_selection - 1) - i][(c_selection - 1) - i] = (char) ( ((int)'0') + nearbymines8);
            i++;
        }
    }
}

```

```

if(c_selection != 1)
{
    i = 0;
    while(nearbymines4 == 0 && (c_selection - i) > 0)
    {
        // This is checking elements to the left
        nearbymines4 = check_for_nearby_mines_b(r_selection - 1, (c_selection - 1 - i));
        if(nearbymines4 != -1)
        {
            game_board_b[r_selection - 1][(c_selection - 1) - i] = (char) ( ((int)'0') + nearbymines4 );

```

```
        i++;  
    }  
}  
}  
}
```

```
// Handles a player winning.
```

```
if(check_win_game_b() == TRUE)
```

```
{  
    system("cls");  
    print_fullboard_b();  
    printf("\n\nYou've won the game!! Congrats!!\n\n");  
    play_again_b();  
}
```

```
return 0;
```

```
}
```

```
int play_game_a()
```

```
{  
    int r_selection = 0, c_selection = 0,  
        nearbymines = 0, nearbymines2 = 0,  
        nearbymines3 = 0, nearbymines4 = 0,  
        nearbymines5 = 0, nearbymines6 = 0,  
        nearbymines7 = 0, nearbymines8 = 0,
```

```

    i = 0;

    // Recieves data from the user, first the row number, then the column number...

    // I could think of other ways to do it, but this one seemed easiest.

    do {

        printf("\nMake a selection (ie. row [ENTER] col): \n");

        printf("Row--> ");

        scanf("%d", &r_selection);

        printf("Col--> ");

        scanf("%d", &c_selection);

    } while(r_selection < 1 || r_selection > BOARD_SIZE_a || c_selection < 1 || c_selection >
BOARD_SIZE_a);

    // ^ Checks for any invalid input statements from user.

    check_for_mine_a(r_selection - 1, c_selection - 1);

    if(lost == 1)

        return -1;

    // Checks for nearby mines at every direction from user input location.

    //Assigns that location the number of mines found nearby, updating the board.

    nearbymines = check_for_nearby_mines_a(r_selection - 1, c_selection - 1);

    game_board_a[r_selection - 1][c_selection - 1] = (char)((int)'0' + nearbymines );

    // The following checks for mines nearby elements

```

```

// in the array with no mines.. it's a continuous
// loop until either a mine is found, or we
// reach the end of the array & it cannot be checked any further.

// It also changes the game_board[] with '0' if no mines are found. Very useful piece of code.
// It checks all elements left, right, up, down and all diagonal directions.
// By running a function that checks these same directions.
// A bit much to follow, though. I'm sure there's a much better way. I just don't know it yet.

if(nearbymines == 0)
{
    if(c_selection != BOARD_SIZE_a)
    {
        i = 0;
        while(nearbymines == 0 && (c_selection - 1 + i) < BOARD_SIZE_a)
        {
            // This is checking elements to the right
            nearbymines = check_for_nearby_mines_a(r_selection - 1, (c_selection - 1 + i));
            if(nearbymines != -1)
            {
                game_board_a[r_selection - 1][(c_selection - 1) + i] = (char) ( ((int)'0') + nearbymines );
                i++;
            }
        }
    }

    if(r_selection != 1)

```

```

{
    i = 0;

    while(nearbymines5 == 0 && (c_selection - 1 + i) < BOARD_SIZE_a && (r_selection - 1 - i) > 0)
    {
        // This is checking elements to the diagonal-uright

        nearbymines5 = check_for_nearby_mines_a((r_selection - 1 - i), (c_selection - 1 + i));

        if(nearbymines5 != -1)
        {
            game_board_a[(r_selection - 1) - i][(c_selection - 1) + i] = (char) ( ((int)'0') + nearbymines5);

            i++;
        }
    }
}

if(r_selection != BOARD_SIZE_a)
{
    i = 0;

    while(nearbymines6 == 0 && (r_selection - 1 + i) < BOARD_SIZE_a && (c_selection - 1 + i) <
BOARD_SIZE_a )
    {
        // This is checking elements to the diagonal-drighr

        nearbymines6 = check_for_nearby_mines_a((r_selection - 1 + i), (c_selection - 1 + i));

        if(nearbymines6 != -1)
        {
            game_board_a[(r_selection - 1) + i][(c_selection - 1) + i] = (char) ( ((int)'0') +
nearbymines6);

            i++;

```

```

        }
    }
}

```

```

if(r_selection != BOARD_SIZE_a)
{
    i = 0;

    while(nearbymines2 == 0 && (r_selection - 1 + i) < BOARD_SIZE_a)
    {
        // This is checking elements heading down

        nearbymines2 = check_for_nearby_mines_a((r_selection - 1 + i), c_selection - 1);

        if(nearbymines2 != -1)
        {
            game_board_a[(r_selection - 1) + i][c_selection - 1] = (char) ( ((int)'0') + nearbymines2 );

            i++;
        }
    }

    if(c_selection != BOARD_SIZE_a)
    {
        i = 0;

        while(nearbymines7 == 0 && (r_selection - 1 + i) < BOARD_SIZE_a && (c_selection - 1 - i) > 0)
        {
            // This is checking elements to the diagonal-dleft

```

```

        nearbymines7 = check_for_nearby_mines_a((r_selection - 1 + i), (c_selection - 1 - i));

        if(nearbymines != -1)

        {

            game_board_a[(r_selection - 1) + i][(c_selection - 1) - i] = (char) ( ((int)'0') + nearbymines7);

            i++;

        }

    }

}

```

```

if(r_selection != 1)

{

    i = 0;

    while(nearbymines3 == 0 && (r_selection - i) > 0)

    {

        // This is checking elements heading up

        nearbymines3 = check_for_nearby_mines_a((r_selection - 1 - i), c_selection - 1);

        if(nearbymines3 != -1)

        {

            game_board_a[(r_selection - 1) - i][c_selection - 1] = (char) ( ((int)'0') + nearbymines3 );

            i++;

        }

    }

    if(c_selection != BOARD_SIZE_a)

    {

```

```

while(nearbymines8 == 0 && (c_selection - 1 - i) > 0 && (r_selection - 1 - i) > 0)
{
    // This is checking elements to the diagonal-uleft
    nearbymines8 = check_for_nearby_mines_a((r_selection - 1 - i), (c_selection - 1 - i));
    if(nearbymines8 != -1)
    {
        game_board_a[(r_selection - 1) - i][(c_selection - 1) - i] = (char) ( ((int)'0') + nearbymines8);
        i++;
    }
}
}
}

```

```

if(c_selection != 1)
{
    i = 0;
    while(nearbymines4 == 0 && (c_selection - i) > 0)
    {
        // This is checking elements to the left
        nearbymines4 = check_for_nearby_mines_a(r_selection - 1, (c_selection - 1 - i));
        if(nearbymines4 != -1)
        {
            game_board_a[r_selection - 1][(c_selection - 1) - i] = (char) ( ((int)'0') + nearbymines4 );
            i++;
        }
    }
}

```

```
    }  
    }  
}
```

```
// Handles a player winning.
```

```
if(check_win_game_a() == TRUE)
```

```
{  
    system("cls");  
    print_fullboard_a();  
    printf("\n\nYou've won the game!! Congrats!!\n\n");  
    play_again_a();  
}
```

```
return 0;
```

```
}
```

```
int play_game_p()
```

```
{  
    int r_selection = 0, c_selection = 0,  
        nearbymines = 0, nearbymines2 = 0,  
        nearbymines3 = 0, nearbymines4 = 0,  
        nearbymines5 = 0, nearbymines6 = 0,  
        nearbymines7 = 0, nearbymines8 = 0,  
        i = 0;
```

```
// Recieves data from the user, first the row number, then the column number...
```

```

// I could think of other ways to do it, but this one seemed easiest.

do {

printf("\nMake a selection (ie. row [ENTER] col): \n");

printf("Row--> ");

scanf("%d", &r_selection);

printf("Col--> ");

scanf("%d", &c_selection);


} while(r_selection < 1 || r_selection > BOARD_SIZE_p || c_selection < 1 || c_selection >
BOARD_SIZE_p);

// ^ Checks for any invalid input statements from user.


check_for_mine_p(r_selection - 1, c_selection - 1);


if(lost == 1)

    return -1;

// Checks for nearby mines at every direction from user input location.

//Assigns that location the number of mines found nearby, updating the board.


nearbymines = check_for_nearby_mines_p(r_selection - 1, c_selection - 1);
game_board_p[r_selection - 1][c_selection - 1] = (char)((int)'0' + nearbymines );


// The following checks for mines nearby elements

// in the array with no mines.. it's a continuous

// loop until either a mine is found, or we

```

```
// reach the end of the array & it cannot be checked any further.
```

```
// It also changes the game_board[] with '0' if no mines are found. Very useful piece of code.
```

```
// It checks all elements left, right, up, down and all diagonal directions.
```

```
// By running a function that checks these same directions.
```

```
// A bit much to follow, though. I'm sure there's a much better way. I just don't know it yet.
```

```
if(nearbymines == 0)
```

```
{
```

```
    if(c_selection != BOARD_SIZE_p)
```

```
    {
```

```
        i = 0;
```

```
        while(nearbymines == 0 && (c_selection - 1 + i) < BOARD_SIZE_p)
```

```
        {
```

```
            // This is checking elements to the right
```

```
            nearbymines = check_for_nearby_mines_p(r_selection - 1, (c_selection - 1 + i));
```

```
            if(nearbymines != -1)
```

```
            {
```

```
                game_board_p[r_selection - 1][(c_selection - 1) + i] = (char) ( ((int)'0') + nearbymines );
```

```
                i++;
```

```
            }
```

```
        }
```

```
        if(r_selection != 1)
```

```
        {
```

```
            i = 0;
```

```

while(nearbymines5 == 0 && (c_selection - 1 + i) < BOARD_SIZE_p && (r_selection - 1 - i) > 0)
{
    // This is checking elements to the diagonal-uright
    nearbymines5 = check_for_nearby_mines_p((r_selection - 1 - i), (c_selection - 1 + i));
    if(nearbymines5 != -1)
    {
        game_board_p[(r_selection - 1) - i][(c_selection - 1) + i] = (char) ( ((int)'0') + nearbymines5);
        i++;
    }
}

if(r_selection != BOARD_SIZE_p)
{
    i = 0;

    while(nearbymines6 == 0 && (r_selection - 1 + i) < BOARD_SIZE_p && (c_selection - 1 + i) <
BOARD_SIZE_p )
    {
        // This is checking elements to the diagonal-drighr
        nearbymines6 = check_for_nearby_mines_p((r_selection - 1 + i), (c_selection - 1 + i));
        if(nearbymines6 != -1)
        {
            game_board_p[(r_selection - 1) + i][(c_selection - 1) + i] = (char) ( ((int)'0') +
nearbymines6);
            i++;
        }
    }
}

```

```
    }  
}
```

```
if(r_selection != BOARD_SIZE_p)  
{  
    i = 0;  
    while(nearbymines2 == 0 && (r_selection - 1 + i) < BOARD_SIZE_p)  
    {  
        // This is checking elements heading down  
        nearbymines2 = check_for_nearby_mines_p((r_selection - 1 + i), c_selection - 1);  
        if(nearbymines2 != -1)  
        {  
            game_board_p[(r_selection - 1) + i][c_selection - 1] = (char) ( ((int)'0') + nearbymines2 );  
            i++;  
        }  
    }  
}  
  
if(c_selection != BOARD_SIZE_p)  
{  
    i = 0;  
    while(nearbymines7 == 0 && (r_selection - 1 + i) < BOARD_SIZE_p && (c_selection - 1 - i) > 0)  
    {  
        // This is checking elements to the diagonal-dleft  
        nearbymines7 = check_for_nearby_mines_p((r_selection - 1 + i), (c_selection - 1 - i));  
        if(nearbymines7 != -1)
```

```

    {
        game_board_p[(r_selection - 1) + i][(c_selection - 1) - i] = (char) ( ((int)'0') + nearbymines7);
        i++;
    }
}
}
}

```

```

if(r_selection != 1)
{
    i = 0;
    while(nearbymines3 == 0 && (r_selection - i) > 0)
    {
        // This is checking elements heading up
        nearbymines3 = check_for_nearby_mines_p((r_selection - 1 - i), c_selection - 1);
        if(nearbymines3 != -1)
        {
            game_board_p[(r_selection - 1) - i][c_selection - 1] = (char) ( ((int)'0') + nearbymines3 );
            i++;
        }
    }
    if(c_selection != BOARD_SIZE_p)
    {
        while(nearbymines8 == 0 && (c_selection - 1 - i) > 0 && (r_selection - 1 - i) > 0)
        {

```

```

// This is checking elements to the diagonal-uleft

nearbymines8 = check_for_nearby_mines_p((r_selection - 1 - i), (c_selection - 1 - i));

if(nearbymines8 != -1)

{

game_board_p[(r_selection - 1) - i][(c_selection - 1) - i] = (char) ( ((int)'0') + nearbymines8);

i++;

}

}

}

}

if(c_selection != 1)

{

i = 0;

while(nearbymines4 == 0 && (c_selection - i) > 0)

{

// This is checking elements to the left

nearbymines4 = check_for_nearby_mines_p(r_selection - 1, (c_selection - 1 - i));

if(nearbymines4 != -1)

{

game_board_p[r_selection - 1][(c_selection - 1) - i] = (char) ( ((int)'0') + nearbymines4 );

i++;

}

}

}

}

```

```

}

// Handles a player winning.
if(check_win_game_p() == TRUE)
{
    system("cls");

    print_fullboard_p();

    printf("\n\nYou've won the game!! Congrats!!\n\n");

    play_again_p();
}

return 0;
}

/* Check whether user input has selected a mine */
void check_for_mine_b(int r_select, int c_select)
{
    if(board_b[r_select][c_select] == '*')
    {
        printf("\nYou've hit a mine! You lose!\n");

        getchar(); getchar();

        lost = 1;
    }
}
}

```

```

void check_for_mine_a(int r_select, int c_select)
{
    if(board_a[r_select][c_select] == '*')
    {
        printf("\nYou've hit a mine! You lose!\n");
        getchar(); getchar();

        lost = 1;
    }
}

```

```

void check_for_mine_p(int r_select, int c_select)
{
    if(board_p[r_select][c_select] == '*')
    {
        printf("\nYou've hit a mine! You lose!\n");
        getchar(); getchar();

        lost = 1;
    }
}

```

//Another ridiculous function to find nearby mines.

//I know, I know...it's messy, and needs a rewrite.

```

int check_for_nearby_mines_b(int r_select, int c_select)
{
    int nearby_mine_count = 0;

```

```

if(board_b[r_select][c_select] == '*')

    return -1;


// Check for mines below and to the right.

if(r_select < (BOARD_SIZE_b - 1) && c_select < (BOARD_SIZE_b - 1))
{
    // Check for mine below

    if(board_b[r_select + 1][c_select] == '*')

        nearby_mine_count++;

    // Check for mine to the right.

    if(board_b[r_select][c_select + 1] == '*')

        nearby_mine_count++;

    // Check for mine diagonal-dright.

    if(board_b[r_select + 1][c_select + 1] == '*')

        nearby_mine_count++;


// Check whether the columns to the left can be checked

if(c_select != 0)

{
    // Check for mine diagonal-dleft

    if(board_b[r_select + 1][c_select - 1] == '*')

        nearby_mine_count++;

    // Check for mine to the left

    if(board_b[r_select][c_select - 1] == '*')

        nearby_mine_count++;
}

```

```

}

// Check whether the rows above can be checked

if(r_select != 0)
{
    // Check for mine above

    if(board_b[r_select - 1][c_select] == '*')

        nearby_mine_count++;

    // Check for mine diagonal-uright

    if(board_b[r_select - 1][c_select + 1] == '*')

        nearby_mine_count++;

    // Check whether columns to the left can be checked

    if(c_select != 0)

    {

        // Check for mine diagonal-uleft

        if(board_b[r_select - 1][c_select - 1] == '*')

            nearby_mine_count++;

    }

}

}

// Check if selection is in last row

if(r_select == (BOARD_SIZE_b - 1) && c_select != (BOARD_SIZE_b - 1))

{

    // Check for mine above

    if(board_b[r_select - 1][c_select] == '*')

        nearby_mine_count++;

```

```

// Check for mine diagonal-uright
    if(board_b[r_select - 1][c_select + 1] == '*')
        nearby_mine_count++;
}

// Check if selection is in last column
if(c_select == (BOARD_SIZE_b - 1) && r_select != (BOARD_SIZE_b - 1))
{
    // Check for mine to the left
    if(board_b[r_select][c_select - 1] == '*')
        nearby_mine_count++;

    // Check for mine diagonal-dleft
    if(board_b[r_select + 1][c_select - 1] == '*')
        nearby_mine_count++;
}

// Check whether selection is last in element
if(r_select == (BOARD_SIZE_b - 1) && c_select == (BOARD_SIZE_b - 1))
{
    // Check for mine to the left
    if(board_b[r_select][c_select - 1] == '*')
        nearby_mine_count++;

    // Check for mine diagonal-dleft
    if(board_b[r_select - 1][c_select - 1] == '*')
        nearby_mine_count++;

    // Check for mine above
    if(board_b[r_select - 1][c_select] == '*')

```

```
        nearby_mine_count++;
    }
    return nearby_mine_count;
}
```

```
int check_for_nearby_mines_a(int r_select, int c_select)
```

```
{
    int nearby_mine_count = 0;
    if(board_a[r_select][c_select] == '*')
        return -1;

    // Check for mines below and to the right.
    if(r_select < (BOARD_SIZE_a - 1) && c_select < (BOARD_SIZE_a - 1))
    {
        // Check for mine below
        if(board_a[r_select + 1][c_select] == '*')
            nearby_mine_count++;

        // Check for mine to the right.
        if(board_a[r_select][c_select + 1] == '*')
            nearby_mine_count++;

        // Check for mine diagonal-dright.
        if(board_a[r_select + 1][c_select + 1] == '*')
            nearby_mine_count++;

        // Check whether the columns to the left can be checked
```

```

if(c_select != 0)
{
    // Check for mine diagonal-dleft
    if(board_a[r_select + 1][c_select - 1] == '*')
        nearby_mine_count++;

    // Check for mine to the left
    if(board_a[r_select][c_select - 1] == '*')
        nearby_mine_count++;
}

// Check whether the rows above can be checked
if(r_select != 0)
{
    // Check for mine above
    if(board_a[r_select - 1][c_select] == '*')
        nearby_mine_count++;

    // Check for mine diagonal-uright
    if(board_a[r_select - 1][c_select + 1] == '*')
        nearby_mine_count++;

    // Check whether columns to the left can be checked
    if(c_select != 0)
    {
        // Check for mine diagonal-uleft
        if(board_a[r_select - 1][c_select - 1] == '*')
            nearby_mine_count++;
    }
}

```

```

    }
}

// Check if selection is in last row
if(r_select == (BOARD_SIZE_a - 1) && c_select != (BOARD_SIZE_a - 1))
{
    // Check for mine above
    if(board_a[r_select - 1][c_select] == '*')
        nearby_mine_count++;

    // Check for mine diagonal-uright
    if(board_a[r_select - 1][c_select + 1] == '*')
        nearby_mine_count++;
}

// Check if selection is in last column
if(c_select == (BOARD_SIZE_a - 1) && r_select != (BOARD_SIZE_a - 1))
{
    // Check for mine to the left
    if(board_a[r_select][c_select - 1] == '*')
        nearby_mine_count++;

    // Check for mine diagonal-dleft
    if(board_a[r_select + 1][c_select - 1] == '*')
        nearby_mine_count++;
}

// Check whether selection is last in element
if(r_select == (BOARD_SIZE_a - 1) && c_select == (BOARD_SIZE_a - 1))
{

```

```

// Check for mine to the left
    if(board_a[r_select][c_select - 1] == '*')
        nearby_mine_count++;

// Check for mine diagonal-dleft
    if(board_a[r_select - 1][c_select - 1] == '*')
        nearby_mine_count++;

// Check for mine above
    if(board_a[r_select - 1][c_select] == '*')
        nearby_mine_count++;
}

return nearby_mine_count;
}

int check_for_nearby_mines_p(int r_select, int c_select)
{
    int nearby_mine_count = 0;

    if(board_p[r_select][c_select] == '*')
        return -1;

// Check for mines below and to the right.
    if(r_select < (BOARD_SIZE_p - 1) && c_select < (BOARD_SIZE_p - 1))
    {
        // Check for mine below
        if(board_p[r_select + 1][c_select] == '*')

```

```

    nearby_mine_count++;

// Check for mine to the right.
if(board_p[r_select][c_select + 1] == '*')

    nearby_mine_count++;

// Check for mine diagonal-dright.
if(board_p[r_select + 1][c_select + 1] == '*')

    nearby_mine_count++;


// Check whether the columns to the left can be checked
if(c_select != 0)
{
    // Check for mine diagonal-dleft
    if(board_p[r_select + 1][c_select - 1] == '*')

        nearby_mine_count++;

    // Check for mine to the left
    if(board_p[r_select][c_select - 1] == '*')

        nearby_mine_count++;
}

// Check whether the rows above can be checked
if(r_select != 0)
{
    // Check for mine above
    if(board_p[r_select - 1][c_select] == '*')

        nearby_mine_count++;

    // Check for mine diagonal-uright

```

```

    if(board_p[r_select - 1][c_select + 1] == '*')
        nearby_mine_count++;

    // Check whether columns to the left can be checked
    if(c_select != 0)
    {
        // Check for mine diagonal-ulleft
        if(board_p[r_select - 1][c_select - 1] == '*')
            nearby_mine_count++;
    }
}

// Check if selection is in last row
if(r_select == (BOARD_SIZE_p - 1) && c_select != (BOARD_SIZE_p - 1))
{
    // Check for mine above
    if(board_p[r_select - 1][c_select] == '*')
        nearby_mine_count++;

    // Check for mine diagonal-uright
    if(board_p[r_select - 1][c_select + 1] == '*')
        nearby_mine_count++;
}

// Check if selection is in last column
if(c_select == (BOARD_SIZE_p - 1) && r_select != (BOARD_SIZE_p - 1))
{
    // Check for mine to the left

```

```

        if(board_p[r_select][c_select - 1] == '*')

            nearby_mine_count++;

// Check for mine diagonal-dleft

        if(board_p[r_select + 1][c_select - 1] == '*')

            nearby_mine_count++;

    }

// Check whether selection is last in element

if(r_select == (BOARD_SIZE_p - 1) && c_select == (BOARD_SIZE_p - 1))

{

    // Check for mine to the left

        if(board_p[r_select][c_select - 1] == '*')

            nearby_mine_count++;

// Check for mine diagonal-dleft

        if(board_p[r_select - 1][c_select - 1] == '*')

            nearby_mine_count++;

// Check for mine above

        if(board_p[r_select - 1][c_select] == '*')

            nearby_mine_count++;

    }

return nearby_mine_count;

}

/* Check if user has won game */

int check_win_game_b()

```

```
{  
  
    int row, col;  
  
    for(row = 0; row < BOARD_SIZE_b; row++)  
        for(col = 0; col < BOARD_SIZE_b; col++)  
            {  
                if(game_board_b[row][col] == 'o' && board_b[row][col] != '*')  
                    return FALSE;  
            }  
  
    return TRUE;  
}
```

```
int check_win_game_a()  
{  
    int row, col;  
  
    for(row = 0; row < BOARD_SIZE_a; row++)  
        for(col = 0; col < BOARD_SIZE_a; col++)  
            {  
                if(game_board_a[row][col] == 'o' && board_a[row][col] != '*')  
                    return FALSE;  
            }  
  
    return TRUE;
```

```
}
```

```
int check_win_game_p()
```

```
{
```

```
    int row, col;
```

```
    for(row = 0; row < BOARD_SIZE_p; row++)
```

```
        for(col = 0; col < BOARD_SIZE_p; col++)
```

```
        {
```

```
            if(game_board_p[row][col] == 'o' && board_p[row][col] != '*')
```

```
                return FALSE;
```

```
        }
```

```
    return TRUE;
```

```
}
```

```
// Ask user if they wish to play again.
```

```
void play_again_b()
```

```
{
```

```
    char ans;
```

```
    printf("\n\n Would You Like To Play Again? \n Press 'Y' or 'y' for YES \n Press 'N' or 'n' for NO \n\n");
```

```
    scanf(" %c", &ans);
```

```
    if(toupper(ans) == 'Y')
```

```
{  
    system("cls");  
    start_b();  
}  
  
else  
{  
    printf("\n\nThanks for playing! Bye.");  
    (void) getchar();  
    exit(EXIT_SUCCESS);  
}  
}  
  
void play_again_a()  
{  
    char ans;  
  
    printf("\n\n Would You Like To Play Again? \n Press 'Y' or 'y' for YES \n Press 'N' or 'n' for NO \n\n");  
    scanf(" %c", &ans);  
  
    if(toupper(ans) == 'Y')  
    {  
        system("cls");  
        start_a();  
    }  
}
```

```
else  
  
{  
  
    printf("\n\nThanks for playing! Bye.");  
  
    (void) getchar();  
  
    exit(EXIT_SUCCESS);  
  
}  
}
```

```
void play_again_p()
```

```
{  
  
    char ans;  
  
  
  
  
    printf("\n\n Would You Like To Play Again? \n Press 'Y' or 'y' for YES \n Press 'N' or 'n' for NO \n\n");  
  
    scanf(" %c", &ans);
```

```
    if(toupper(ans) == 'Y')
```

```
{  
  
    system("cls");  
  
    start_p();  
  
}
```

```
else
```

```
{  
  
    printf("\n\nThanks for playing! Bye.");
```

```

    (void) getchar();

    exit(EXIT_SUCCESS);
}

}

// Displays the welcome message, and the GNU License
void display_welcome()
{
    puts("-----WELCOME TO MINESWEEPER!-----\n\n");
    puts("-----Instructions-----\n");
    puts("1. Type coordinates when prompted");
    puts("2. If you hit a mine, you die");
    puts("3. Do not uncover coordinates you think are mines");
    puts("\n\n");
    puts("-----GOOD LUCK-----\n\n");
}

void start_b()
{
    lost = 0;  // User hasn't lost yet

    // Build both game boards (one for the user to see and the one with the mines).
    build_board_b();
    build_gboard_b();

    // Start playing game

```

```
do

{

play_game_b();

print_board_b();

} while(lost != 1); // While the user hasn't lost, loop.


// Once user is lost, print the board with all the mines.

print_fullboard_b();


// Play again?

play_again_b();

}


void start_a()

{

lost = 0; // User hasn't lost yet

// Build both game boards (one for the user to see and the one with the mines).

build_board_a();

build_gboard_a();


// Start playing game

do

{

play_game_a();

print_board_a();
```

```

    } while (lost != 1); // While the user hasn't lost, loop.

    // Once user is lost, print the board with all the mines.
    print_fullboard_a();

    // Play again?
    play_again_a();
}

void start_p()
{
    lost = 0; // User hasn't lost yet

    // Build both game boards (one for the user to see and the one with the mines).
    build_board_p();
    build_gboard_p();

    // Start playing game
    do
    {
        play_game_p();
        print_board_p();
    } while (lost != 1); // While the user hasn't lost, loop.

    // Once user is lost, print the board with all the mines.
    print_fullboard_p();
}

```

```
// Play again?  
play_again_p();  
}
```