

**IMPLEMENTATION OF CHECKERS**

**Submitted by:**

**Aditya Dan – ENG17CS0009**

**Amruth Reddy – ENG17CS0029**

**Arvin Pradhan – ENG17CS0038**

**Abhishek Singh -ENG17CS0008**

**Anoop M.R. – ENG17CS0035**

of

***Bachelor of Technology***

in

***Computer Science Engineering***

at

***Dayananda Sagar University,***

***School of Engineering, Bangalore-560068***

DATA STRUCTURE AND APPLICATION;

IIIrd semester. (course code: 16CS271)

**Submitted to:**

**Dr. Mallanagouda Patil,**

**Associate Professor, CSE Dept.**

**DAYANANDA SAGAR UNIVERSITY**



**CERTIFICATE**

This is to certify that the DATA STRUCTURE AND APPLICATION Laboratory Mini-Project report entitled **“Implementation of Checkers”** being submitted by

**Aditya Dan (ENG17CS0009), Abhishek Singh (ENG17CS0008), Amruth Reddy (ENG17CS0029), Arvin Pradhan (ENG17CS0038) and Anoop MR (ENG17CS0035)** to Department of Computer Science and Engineering, School of Engineering, Dayananda Sagar University, Bangalore, for the 3rd semester B. Tech C.S.E of this university during the academic year 2018-2019.

***Date***:\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***Signature of the Faculty in Charge***

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***Signature of the Chairman***

**ACKNOWLEDGEMENT**

### We are pleased to acknowledge Dr. Mallanagouda Patil for his invaluable guidance, support, motivation and patience during the course of this mini- project work.

We extend our sincere thanks to our **Chairman Dr. Banga M.K** who continuously helped us throughout the project and without his guidance, this project would have been an uphill task.

We have received a great deal of guidance and co-operation from our friends and we wish to thank one and all that have directly or indirectly helped us in the successful completion of this mini-project work.

**Aditya Dan**

**Abhishek Singh**

**Arvin Pradhan**

**Amruth Reddy**

**Anoop MR**

TABLE OF CONTENTS

**SL.NO** **TITLE**   **Page No**

Cover Page 1

Certificate 2

Acknowledgement 3

Contents 4

1. Problem Statement 5

2. Abstract 5

2.1 Restrictions & Limitations 6

3. Introduction 6

4. S/W & H/W Requirements 7

5. Code 8

6. Expected Output Analysis 12

7. Conclusion 17

8. References 18

**1. PROBLEM STATEMENT**

**Draughts** ([British English](https://en.wikipedia.org/wiki/British_English)) or **checkers** ([American English](https://en.wikipedia.org/wiki/American_English)) is a group of [strategy](https://en.wikipedia.org/wiki/Abstract_strategy_game) [board games](https://en.wikipedia.org/wiki/Board_game) for two players which involve [diagonal](https://en.wikipedia.org/wiki/Diagonal) moves of uniform game pieces and mandatory captures by jumping over opponent pieces.

The aim of the project is to create a simple text-based implementation of player vs. player checkers.

**2. ABSTRACT**

Draughts is played by two opponents, on opposite sides of the game board. One player has the dark pieces; the other has the light pieces. Players alternate turn. A player may not move an opponent's piece. A move consists of moving a piece diagonally to an adjacent unoccupied square. If the adjacent square contains an opponent's piece, and the square immediately beyond it is vacant, the piece may be captured (and removed from the game) by jumping over it.

Only the dark squares of the checkered board are used. A piece may move only diagonally into an unoccupied square. Capturing is mandatory in most official rules, although some rule variations make capturing optional when presented. In almost all variants, the player without pieces remaining, or who cannot move due to being blocked, loses the game.

**2.1 RESTRCTIONS AND LIMITATIONS**

The program is written in such a way that the game pieces can only move diagonally. It cannot jump two diagonal spaces. The crowned pieces cannot capture in the opposite direction of its normal movement. Capturing is not mandatory in this version of the game, and multiple captures are not possible. In the gameplay, the user must enter square name for crown movement, instead of direction. Moreover, when the board is displayed, the square indices are not perfectly spaced. Crowned pieces behave as flying kings, but can capture only normally.

**3. INTRODUCTION**

Checkers, also known as draughts, is a board game played on an 8\*8 board identical to a chessboard. There are 64 squares on the board, 32 of which are dark, 32 of which are light, arranged alternatingly. There are two players, one of whom has dark or black pieces; the other has light or white pieces. There are twenty identical pieces with each player, each of which is placed on all the dark squares of the first four rows of the board from either side. Pieces move diagonally on the black squares. Before a piece has been crowned, it moves one square forward diagonally. The objective of the game is to capture the opponent’s pieces. To capture an opponent’s piece, the two pieces must be kept directly diagonal to each other and the subsequent square should be empty. If a piece reaches the final rank of the board, it is said to be crowned. A crowned piece can move multiple squares forward in either direction. Ultimately the player who captures all of the opponent’s pieces wins.

4. SOFTWARE REQUIREMENTS

The project was coded using CLion, on a Ubuntu system.

The game can be played using GCC as well.

5. CODE

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

/\*

piece is a structure having the members color and crown. color can be either ‘B’ (black) or ‘W’ (white). crown is 0 when the piece is uncrowned and is made to 1 when the piece gets crowned.

\*/

struct piece {

char color;

int crown;

};

/\*

square is the structure representing the squares on the checkerboard. Every square has a designated name of two characters showing its position on the board (e.g. ‘A1’, ‘B3’, etc.) and this name is stored in the variable name. The variables row and col store which row and column of the board the square is in respectively. The variable piece is a pointer to the piece that occupies the square at that given time. The variable active stores 1 for black squares and 0 for white squares (which are not used in the game).

\*/

struct square {

char name[3];

struct piece \*piece;

int row;

int col;

int active;

};

/\*

The structure player represents the two players in the game. The variable color is ‘W’ for player white and ‘B’ for player black. The variable pieces stores the number of pieces remaining for each player. The variable opponent is a pointer to the variable representing the other player.

\*/

struct player {

char color;

int pieces;

struct player \*opponent;

};

typedef struct square \*SQ;

typedef struct piece \*PC;

typedef struct player \*PLAYER;

/\*

The function getName() takes the input of the column and the row that a square is located in and returns a generated name for that square.

\*/

char \*getName(int row, int col) {

char \*name = (char \*) malloc(3);

name[0] = (char) (65 + col);

name[1] = (char) (49 + row);

name[2] = '\0';

return name;

}

/\*

The function createSquare() receives row and column locations of a square and creates an instance of struct square and fills all the data members according to the position.

\*/

SQ createSquare(int row, int col) {

SQ s = (SQ) malloc(sizeof(struct square));

strcpy(s->name, getName(row, col));

s->row = row;

s->col = col;

s->active = (row + col) % 2 == 0;

s->piece = NULL;

return s;

}

SQ board[8][8];

PLAYER white;

PLAYER black;

/\*

The function createBoard() creates each square of the checkerboard in a loop using createSquare() function.

\*/

void createBoard() {

for (int row = 0; row < 8; ++row) {

for (int col = 0; col < 8; ++col) {

board[row][col] = createSquare(row, col);

}

}

}

/\*

The function assignBlackPieces() creates the instances of struct piece that belong to the player black using a loop.

\*/

void assignBlackPieces() {

for (int row = 5; row < 8; ++row) {

for (int col = 0; col < 8; ++col) {

SQ s = board[row][col];

if (s->active) {

PC pc = (PC) malloc(sizeof(struct piece));

pc->color = 'B';

(board[row][col])->piece = pc;

}

}

}

}

/\*

The function assignWhitePieces() creates the instances of struct piece that belong to the player white using a loop.

\*/

void assignWhitePieces() {

for (int row = 0; row < 3; ++row) {

for (int col = 0; col < 8; ++col) {

SQ s = board[row][col];

if (s->active) {

PC pc = (PC) malloc(sizeof(struct piece));

pc->color = 'W';

(board[row][col])->piece = pc;

}

}

}

}

/\*

In the function assignPlayers, the data members of both player black and white are initialised.

\*/

void assignPlayers() {

white = (PLAYER) malloc(sizeof(struct player));

black = (PLAYER) malloc(sizeof(struct player));

white->color = 'W';

white->pieces = 12;

white->opponent = black;

black->color = 'B';

black->pieces = 12;

black->opponent = white;

}

/\*

The function getDestSquare computes and returns the destination square a piece is going towards by taking the player, direction, source square and number of steps as input.

\*/

SQ getDestSquare(PLAYER p, char dir, SQ src, int steps) {

int destRow = -1;

int destCol = -1;

SQ dest = NULL;

if (p->color == 'W') {

destRow = src->row + steps;

if (dir == 'L') {

destCol = src->col - steps;

} else {

destCol = src->col + steps;

}

} else { //BLACK SIDE

destRow = src->row - steps;

if (dir == 'L') {

destCol = src->col + steps;

} else {

destCol = src->col - steps;

}

}

if (destRow > -1 && destRow < 8 && destCol > -1 && destCol < 8) {

dest = board[destRow][destCol];

}

return dest;

}

/\*

The function isSrcOk() takes the player variable and the source square of a piece as inputs and computes whether or not the source square entered by the user is valid. If the source square entered is valid, the function returns 1, else (if the square is empty or the piece on that square does not belong to the player) it returns 0.

\*/

int isSrcOk(PLAYER player, SQ src) {

if (src->piece == NULL) {

printf("Piece is missing\n");

return 0; // BLANK CELL

}

if (player->color != src->piece->color) {

printf("Piece does not belong to player\n");

return 0; // NOT PLAYERS PIECE

}

return 1;

}

/\*

The function isDestOk() takes a square as input and returns 1 if the square is empty and if it is a valid square on the checkerboard, else it returns 0.

\*/

int isDestOk(SQ dest) {

if (dest == NULL) {

printf("Square is off board\n");

return 0; //DESTINATION DOES NOT EXIST ON BOARD;

}

if (dest->piece != NULL) {

printf("Destination is occupied\n");

return 0; // NOT EMPTY SQUARE

}

return 1;

}

/\*

The function isCrown() takes a player and the square of a piece as inputs and returns 1 if the square is in the final rank of the board (i.e. row 7 for white and row 0 for black).

\*/

int isCrown(PLAYER p, SQ sq) {

return (p->color == 'W' && sq->row == 7) || (p->color == 'B' && sq->row == 0);

}

/\*

The function move() takes player variable, source square and direction as inputs, and computes the destination square. If the destination square exists and is unoccupied, the square of the piece at source square is updated and 1 is returned, else 0 is returned.

\*/

int move(PLAYER player, SQ src, char dir) {

if (!isSrcOk(player, src)) {

return 0;

}

SQ dest = getDestSquare(player, dir, src, 1);

if (!isDestOk(dest)) {

return 0;

}

PC pc = src->piece;

src->piece = NULL;

dest->piece = pc;

if (isCrown(player, dest)) {

pc->crown = 1;

}

return 1;

}

/\*

The function isCapturable() takes the opponent player and the square of a piece as inputs, and returns 1 if the piece can be captured by the player, else returns 0.

\*/

int isCapturable(PLAYER p, SQ sq) {

return (sq->piece != NULL && sq->piece->color != p->color);

}

int capture(PLAYER player, SQ src, char dir) {

if (!isSrcOk(player, src)) {

return 0;

}

SQ cap = getDestSquare(player, dir, src, 1);

if (!isCapturable(player, cap)) {

printf("Square cannot be captured\n");

return 0;

}

SQ dest = getDestSquare(player, dir, src, 2);

if (!isDestOk(dest)) {

return 0;

}

cap->piece = NULL;

--player->opponent->pieces;

PC pc = src->piece;

src->piece = NULL;

dest->piece = pc;

if (isCrown(player, dest)) {

pc->crown = 1;

}

return 1;

}

/\*

The function moveCrown() takes a player variable, source square and destination square as inputs, and computes whether or not the destination square is located diagonal to the source square. If yes, the function further checks if the move is a simple capture move. If yes, capture() function is called, the source’s piece is made to NULL, and destination piece is updated to the piece being moved. In both the cases, 1 is returned. If the diagonality relation of the squares is not satisfied, 0 is returned.

\*/

int moveCrown(PLAYER p, SQ src, SQ dest) {

int vsteps = dest->row - src->row;

int hsteps = dest->col - src->col;

if (vsteps != hsteps) {

printf("Move is not in diagonal path\n");

return 0;

}

if (abs(vsteps) == 2) {

if (p->color == 'W' && vsteps > 0) {

char dir = (char) (hsteps > 0 ? 'R' : 'L');

if (capture(p, src, dir)) {

return 1;

}

}

if (p->color == 'B' && vsteps < 0) {

char dir = (char) (hsteps > 0 ? 'L' : 'R');

if (capture(p, src, dir)) {

return 1;

}

}

}

PC pc = src->piece;

src->piece = NULL;

dest->piece = pc;

return 1;

}

/\*

The function viewBoard() provides an image-like representation of the checkerboard by printing ‘[ ]’ for active empty squares, ‘[W]’ for squares occupied by white pieces, and ‘[B]’ for squares occupied by black pieces.

\*/

void viewBoard() {

for (int row = 7; row >= 0; --row) {

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

for (int col = 0; col < 8; ++col) {

SQ sq = board[row][col];

if (sq->active) {

if (sq->piece != NULL) {

printf("[%c]", sq->piece->crown ? toupper(sq->piece->color) : tolower(sq->piece->color));

} else {

printf("[ ]");

}

} else {

printf(" ");

}

}

printf("\n");

}

printf("---------------------\n");

printf(" A B C D E F G H\n");

}

/\*

The function checkWinnerIfAny() returns player black if the number of pieces of player white becomes 0, and vice-versa.

\*/

PLAYER checkWinnerIfAny(){

if(white->pieces == 0) return black;

if(black->pieces == 0) return white;

return NULL;

}

/\*

The function findSquare() takes the designated name of a square as input and returns the corresponding square in the array board by matching the name of the square to the name entered.

\*/

SQ findSquare(char \*name) {

for (int row = 0; row < 8; ++row) {

for (int col = 0; col < 8; ++col) {

SQ sq = board[row][col];

if (!strcmp(sq->name, name) && sq->active) {

return sq;

}

}

}

return NULL;

}

void initialise() {

createBoard();

assignBlackPieces();

assignWhitePieces();

assignPlayers();

}

void upperCase(char str[]) {

int c = 0;

while (str[c] != '\0') {

str[c] = (char) toupper(str[c]);

c++;

}

}

int main() {

initialise();

viewBoard();

char sq[3], dir,moveType;

PLAYER player, winner;

SQ srcSq = NULL;

SQ destSq = NULL;

FILE \*log;

int turn = 0, success = 0;

if ((log = fopen("log.txt", "a")) == NULL) {

printf("cannot open log file\n");

exit(1);

}

fprintf(log,"-------New Game Starts----------\n");

fclose(log);

while (1) {

input:

moveType = 'X';

dir = 'X';

strcpy(sq, "XX");

srcSq = NULL, destSq = NULL;

player = turn ? black : white;

printf("%c TO MOVE\n", toupper(player->color));

while (moveType != '1' && moveType != '2' && moveType != '3') {

printf("Select Move (1 for Move, 2 for Capture, 3 for Crown Move)\n");

scanf(" %c", &moveType);

}

while (srcSq == NULL || !isSrcOk(player, srcSq)) {

printf("Enter SQUARE [e.g. A1, D8]\n");

scanf("%s", sq);

upperCase(sq);

srcSq = findSquare(sq);

if (moveType == '3' && !srcSq->piece->crown) {

printf("Not a Crown piece\n");

srcSq = NULL;

goto input;

}

}

if (moveType == '3') {

while (destSq == NULL || !isDestOk(destSq)) {

printf("Enter DESTINATION SQUARE [e.g. A1, D8]\n");

scanf("%s", sq);

upperCase(sq);

destSq = findSquare(sq);

}

} else {

while(toupper(dir) != 'L' && toupper(dir) != 'R'){

printf("Enter DIRECTION [either L or R] \n");

scanf(" %c", &dir);

}

}

switch (moveType) {

case '1':

success = move(player, srcSq, (char) toupper(dir));

break;

case '2':

success = capture(player, srcSq, (char) toupper(dir));

break;

case '3':

success = moveCrown(player,srcSq,destSq);

break;

}

if (success) {

log = fopen("log.txt","a");

if(moveType == '3'){

fprintf(log, "Player %c : Move %c : Src Square %s : Dest Square %s\n", player->color, moveType, srcSq->name, destSq->name);

}else{

fprintf(log, "Player %c : Move %c : Src Square %s : Direction %c\n", player->color, moveType, srcSq->name, dir);

}

fclose(log);

turn = !turn;

viewBoard();

winner = checkWinnerIfAny();

if(winner != NULL){

printf("Player %c wins !!\n",winner->color);

break;

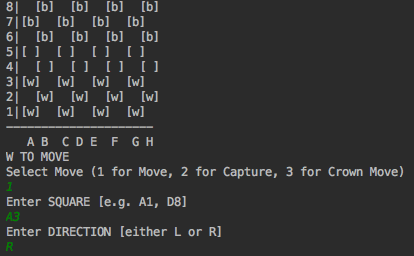
}

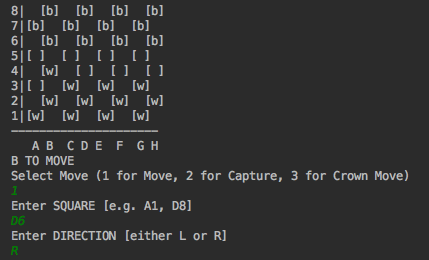
}

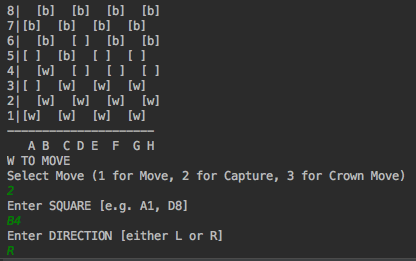
}

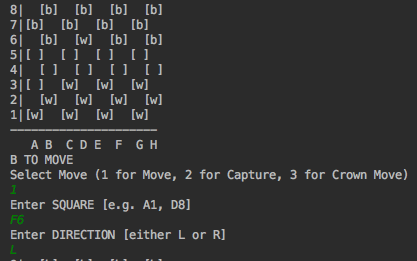
}

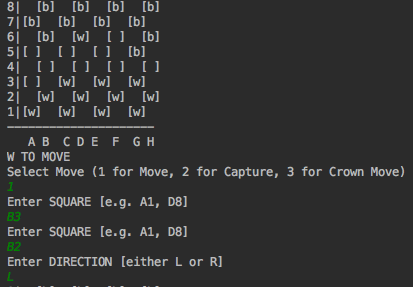
6. OUTPUT

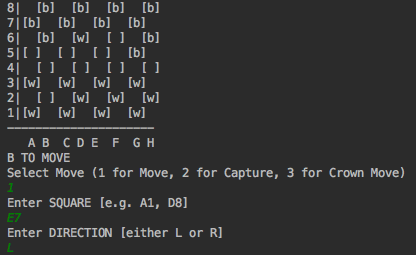


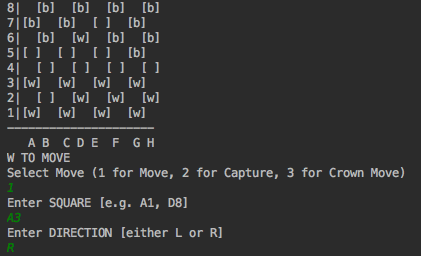


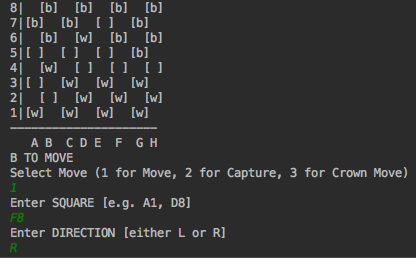


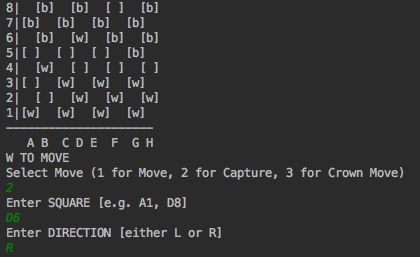


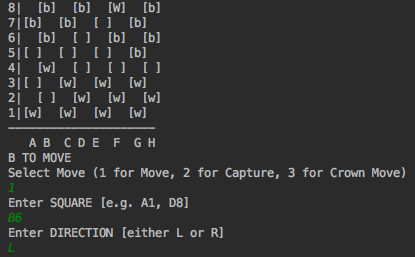


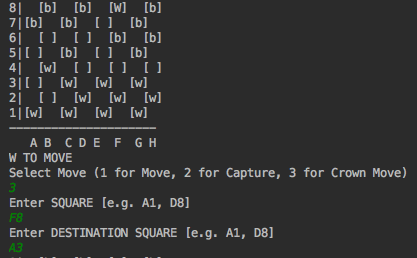


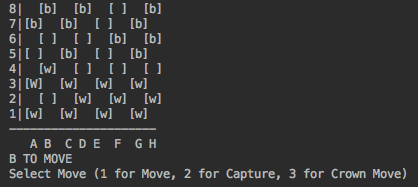












7. CONCLUSION.

A simple game of checkers has been implemented on a text-based format using C language. This game requires two players who take turns to enter their move in the game. In future this game can be improvised and be made into a GUI-based game.

**8. REFERENCES**

**[1]** [**https://github.com/teamb58/**](https://github.com/teamb58/Hangman)

**[2]** [**www.wikipedia.org**](http://www.wikipedia.org/)

**[3]** [**www.cse.iitb.ac.in**](http://www.cse.iitb.ac.in/)

**[4]** [**www.codeproject.com**](http://www.codeproject.com/)