Fundamentals of computer programming

Bisma Ijaz

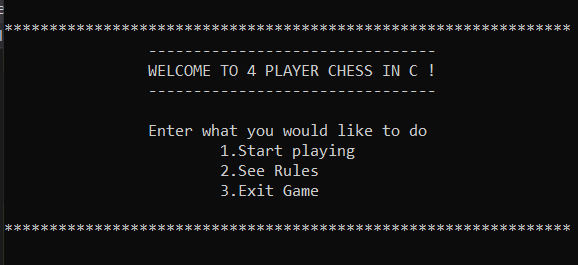
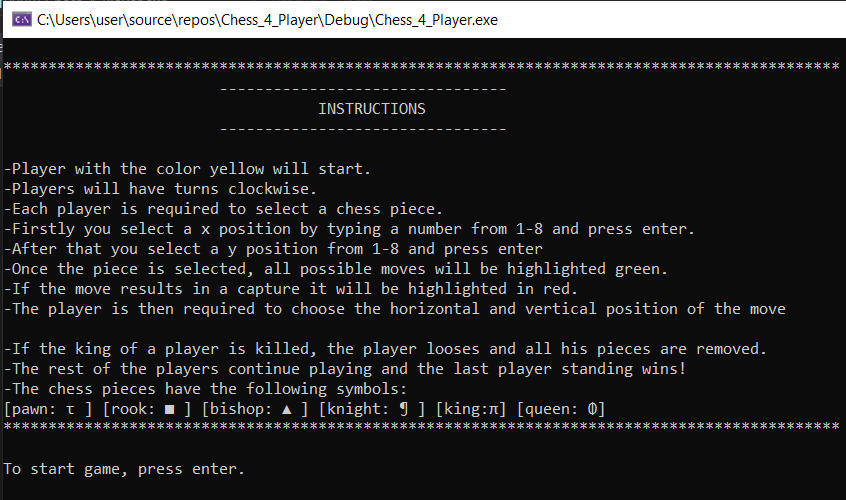
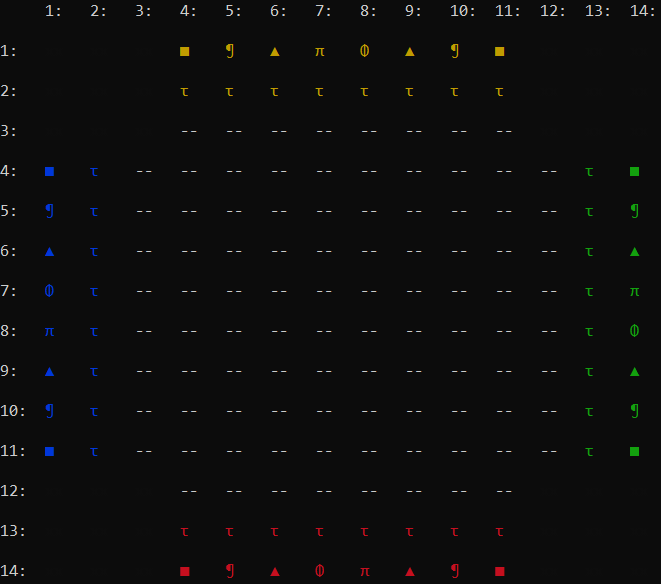
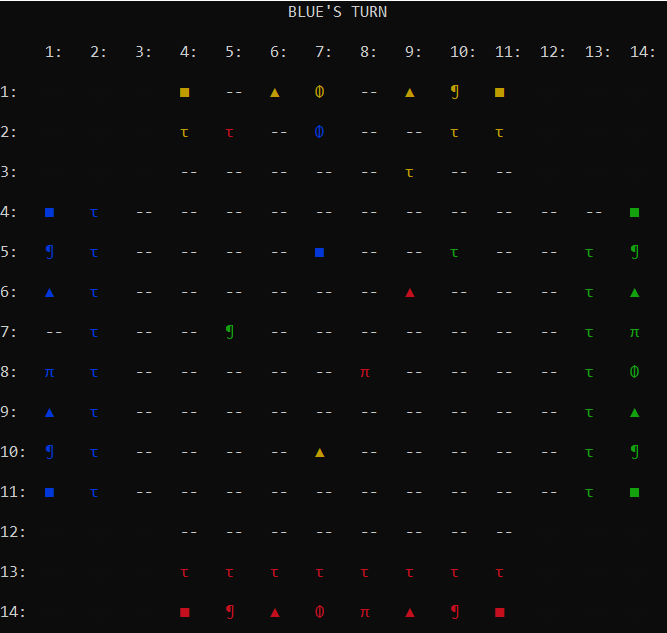
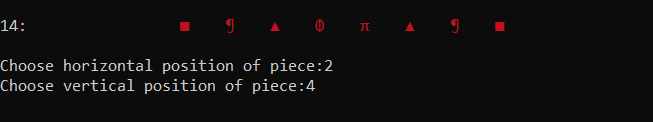
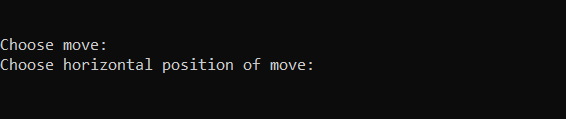
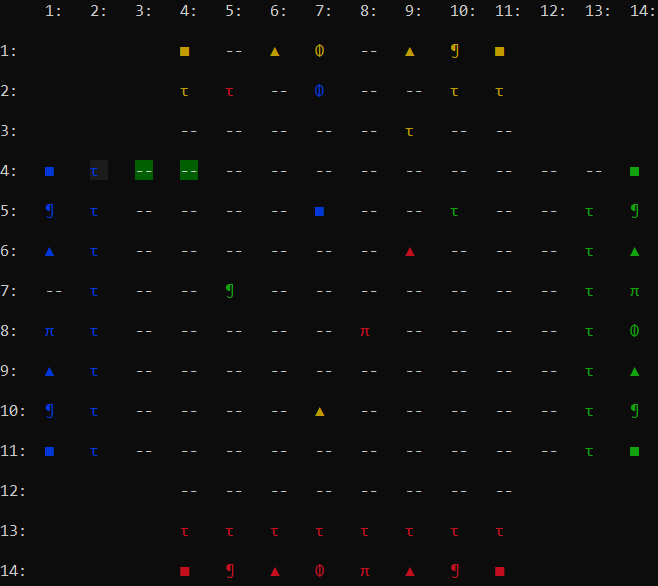
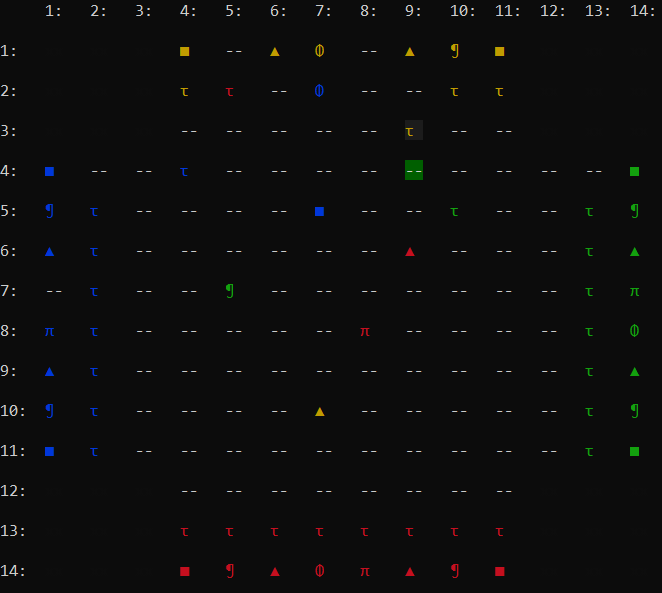
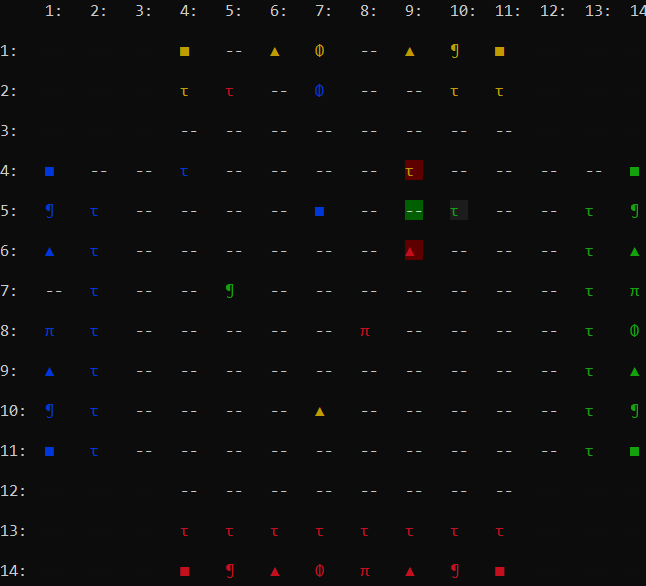
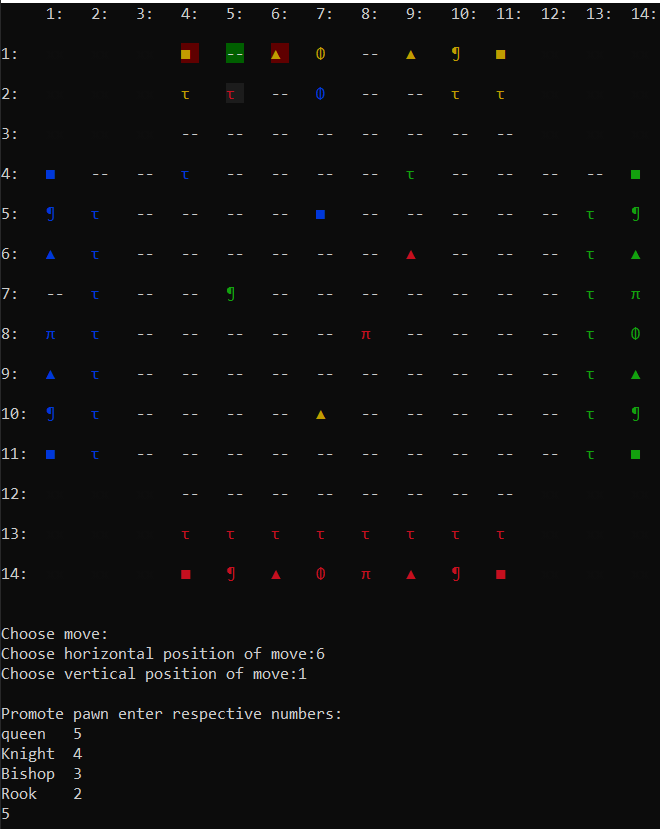
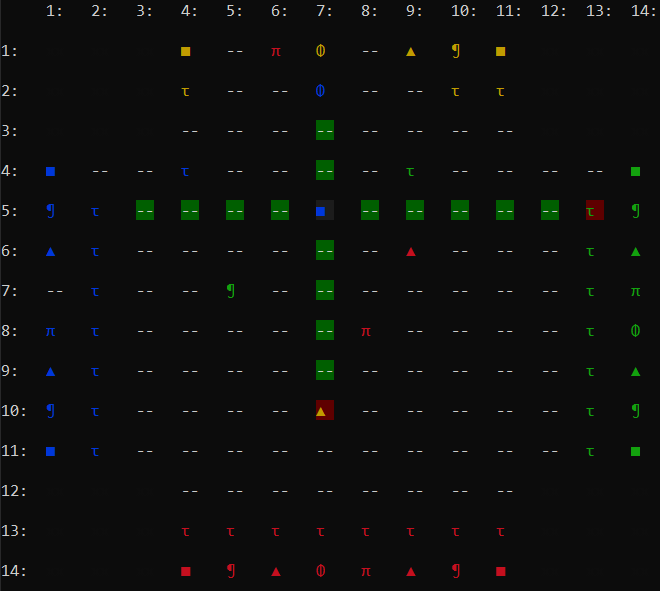
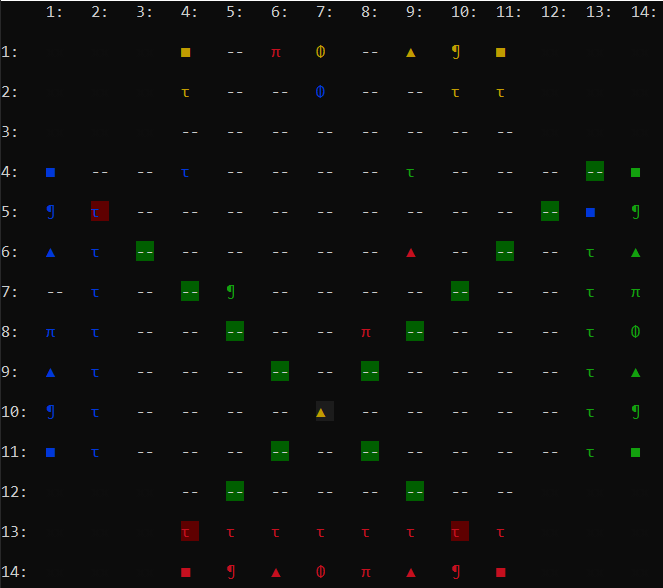
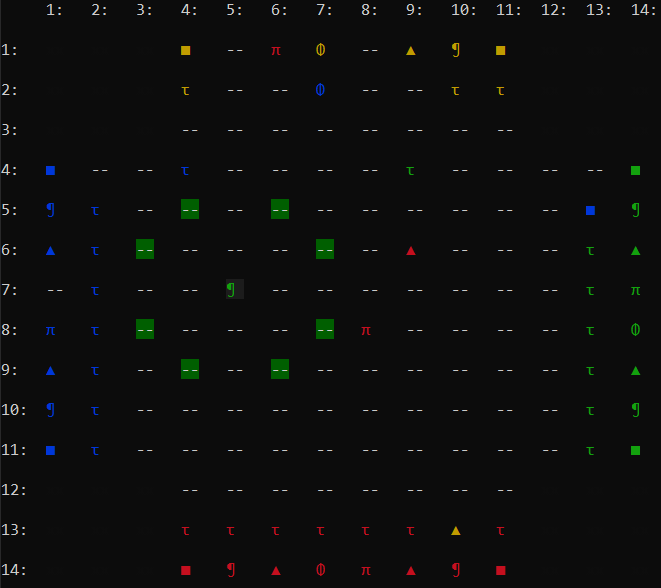
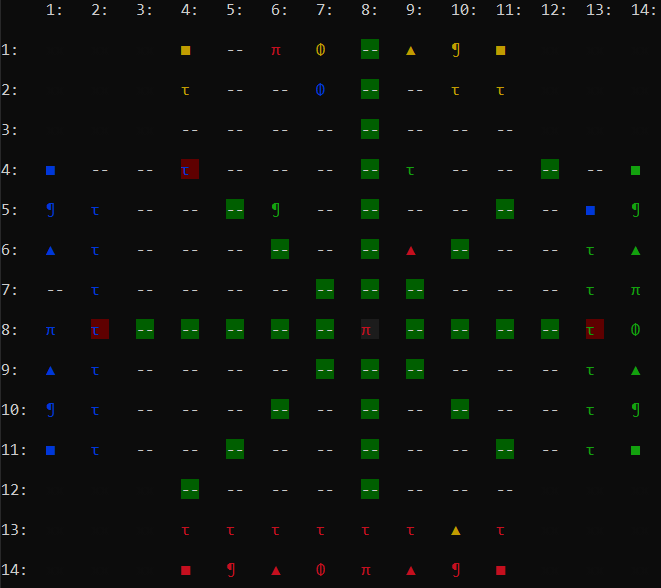
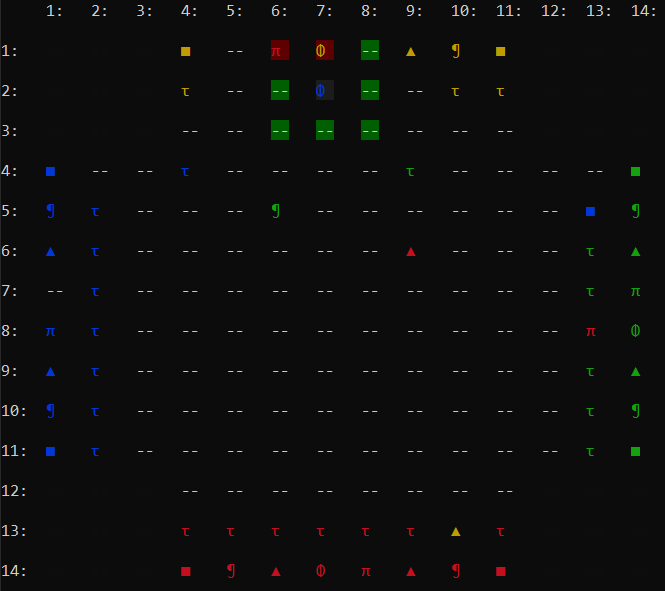
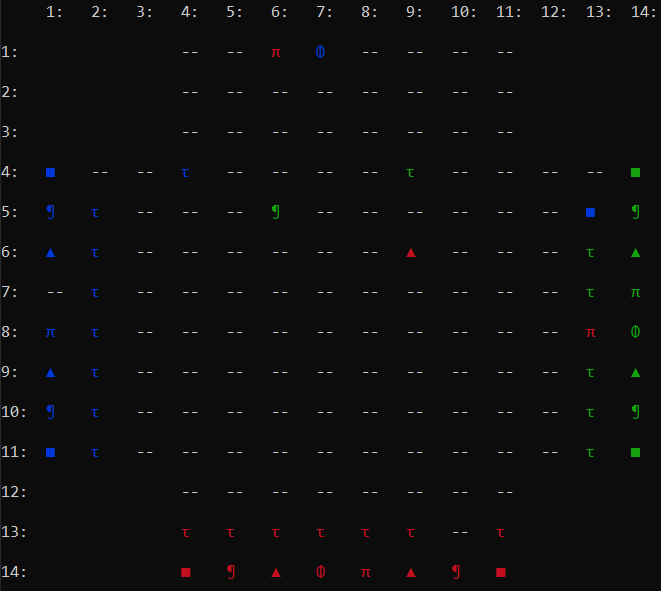
Project: 4 Player Chess Game

1. Note: the vision for this game, basic idea, functionality of different pieces and the basic funcitonality of the programme itself can be found in the following file(excel file contained within the zip file): <PROJECT.xls>
2. The major following Arrays were initialized in the program:

|  |  |
| --- | --- |
| Arrays | Reason/Use |
| Board\_Pieces[14][14] | * Stores each piece with its unique number |
| Player\_Colors[14][14] | * Stores colors of four players |
| Moves\_Array[14][14] | * Stores valid moves and capture moves. |

1. The following functions were introduced in the program:

|  |  |
| --- | --- |
| Functions | Reason/Use |
| void background\_color(int); | * Sets background color of board:   + Red = move that ends in capturing another piece.   + Green= valid move.   + Grey= the piece that’s moving.   + Black=rest of the board. |
| int piece\_symbols(int); | * Determines the unique symbol for each piece |
| void Chess\_Board(void); | * Determines the color of that piece and prints the final board. |
| int Valid\_Piece\_Moves(int, int); | * Checks for valid moves of the piece being used by the current player. |

1. Input code:
2. #include<stdio.h>
3. #include<math.h>
4. #include<ctype.h>
5. #include<windows.h>
6. //defining colors
7. #define yellow 1
8. #define green 2
9. #define red 3
10. #define blue 4
11. int Current\_player = blue;
12. //defining pieces//
13. #define pawn 1
14. #define rook 2
15. #define bishop 3
16. #define knight 4
17. #define queen 5
18. #define king 6
19. //declaring functions
20. void Chess\_Board(void);
21. int Valid\_Piece\_Moves(int, int);
22. int piece\_symbols(int);
23. void background\_color(int);
24. //decalring other variables used in the program
25. int flag = 0;
26. int num = -1;
27. int lost\_player[3] = { 0 };
28. //orignal board
29. int Board\_Pieces[14][14] =
30. {
31. {9,9,9,2,4,3,5,6,3,4,2,9,9,9},
32. {9,9,9,1,1,1,1,1,1,1,1,9,9,9},
33. {9,9,9,0,0,0,0,0,0,0,0,9,9,9},
34. {2,1,0,0,0,0,0,0,0,0,0,0,1,2},
35. {4,1,0,0,0,0,0,0,0,0,0,0,1,4},
36. {3,1,0,0,0,0,0,0,0,0,0,0,1,3},
37. {6,1,0,0,0,0,0,0,0,0,0,0,1,5},
38. {5,1,0,0,0,0,0,0,0,0,0,0,1,6},
39. {3,1,0,0,0,0,0,0,0,0,0,0,1,3},
40. {4,1,0,0,0,0,0,0,0,0,0,0,1,4},
41. {2,1,0,0,0,0,0,0,0,0,0,0,1,2},
42. {9,9,9,0,0,0,0,0,0,0,0,9,9,9},
43. {9,9,9,1,1,1,1,1,1,1,1,9,9,9},
44. {9,9,9,2,4,3,6,5,3,4,2,9,9,9}
45. };
46. int Player\_Colors[14][14] =
47. {
48. {9,9,9,1,1,1,1,1,1,1,1,9,9,9},
49. {9,9,9,1,1,1,1,1,1,1,1,9,9,9},
50. {9,9,9,0,0,0,0,0,0,0,0,9,9,9},
51. {4,4,0,0,0,0,0,0,0,0,0,0,2,2},
52. {4,4,0,0,0,0,0,0,0,0,0,0,2,2},
53. {4,4,0,0,0,0,0,0,0,0,0,0,2,2},
54. {4,4,0,0,0,0,0,0,0,0,0,0,2,2},
55. {4,4,0,0,0,0,0,0,0,0,0,0,2,2},
56. {4,4,0,0,0,0,0,0,0,0,0,0,2,2},
57. {4,4,0,0,0,0,0,0,0,0,0,0,2,2},
58. {4,4,0,0,0,0,0,0,0,0,0,0,2,2},
59. {9,9,9,0,0,0,0,0,0,0,0,9,9,9},
60. {9,9,9,3,3,3,3,3,3,3,3,9,9,9},
61. {9,9,9,3,3,3,3,3,3,3,3,9,9,9}
62. };
63. int Moves\_Array[14][14] = {0};
64. //start of program
65. void main()
66. {
67. //start of program
68. while (1) {
69. system("@cls||clear");
70. printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");
71. printf("\t\t--------------------------------\n");
72. printf("\t\tWELCOME TO 4 PLAYER CHESS IN C !\n");
73. printf("\t\t--------------------------------\n\n");
74. printf("\t\tEnter what you would like to do\n");
75. printf("\t\t\t1.Start playing\n");
76. printf("\t\t\t2.See Rules\n");
77. printf("\t\t\t3.Exit Game\n");
78. printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");
79. int tab;
80. scanf\_s("%d", &tab);
81. printf("\n\n");
82. switch (tab) {
83. case 1://game begins
84. goto start;
85. break;
87. case 2://rules
88. system("@cls||clear");
89. printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");
90. printf("\t\t\t--------------------------------\n");
91. printf("\t\t\t INSTRUCTIONS\n");
92. printf("\t\t\t--------------------------------\n\n");
93. printf("-Player with the color yellow will start.\n");
94. printf("-Players will have turns clockwise.\n");
95. printf("-Each player is required to select a chess piece.\n");
96. printf("-Firstly you select a x position by typing a number from 1-8 and press enter.\n");
97. printf("-After that you select a y position from 1-8 and press enter\n");
98. printf("-Once the piece is selected, all possible moves will be highlighted green.\n");
99. printf("-If the move results in a capture it will be highlighted in red.\n");
100. printf("-The player is then required to choose the horizontal and vertical position of the move\n\n");
101. printf("-If the king of a player is killed, the player looses and all his pieces are removed.\n");
102. printf("-The rest of the players continue playing and the last player standing wins!\n");
103. printf("-The chess pieces have the following symbols:\n[pawn: %c ] [rook: %c ] [bishop: %c ] [knight: %c ] [king:%c] [queen: %c]", 231, 254, 30, 20, 227, 232);
104. printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");
105. printf("\nTo go back and start game, press any number and enter.");
106. scanf\_s("%d", &tab);
107. break;
108. case 3://game ends
109. printf("Thank you for playing!");
110. return 0;
111. break;
112. }
113. }
114. start:
115. while (1)
116. {
117. system("@cls||clear");
118. switch (Current\_player)//printing the turn of player
119. {
120. case yellow: printf("\t\t\t\tYELLOW'S TURN\n\n"); break;
121. case green: printf("\t\t\t\tGREEEN'S TURN\n\n"); break;
122. case red: printf("\t\t\t\tRED'S TURN\n\n"); break;
123. case blue: printf("\t\t\t\tBLUE'S TURN\n\n"); break;
124. }
125. Chess\_Board();//board printed
126. //player chooses which piece to move
127. int a;
128. printf("Choose horizontal position of piece:");
129. scanf\_s("%d", &a);
130. int x = a - 1;
131. printf("Choose vertical position of piece:");
132. scanf\_s("%d", &a);
133. int y = a - 1;
134. if (Board\_Pieces[y][x] == 0 || Board\_Pieces[y][x]==9)//if piece chose if off board or none
135. continue;
136. if (Player\_Colors[y][x] != Current\_player)//if piece chosen is not of the player
137. continue;
138. if (Valid\_Piece\_Moves(x, y) == 1)//no valid moves hence loop continued
139. {
140. for (int x\_ = 0; x\_ < 14; x\_++) //resetting move array to zero.
141. for (int y\_ = 0; y\_ < 14; y\_++)
142. Moves\_Array[x\_][y\_] = 0;
143. continue;
144. }
145. system("@cls||clear");
146. Chess\_Board();// moves shown
147. printf("\n");
148. //inputting move
149. int input[2];
150. int c;
151. printf("Choose move:\n");
152. printf("Choose horizontal position of move:");
153. scanf\_s("%d", &c);
154. input[0] = c - 1;
155. printf("Choose vertical position of move:");
156. scanf\_s("%d", &c);
157. input[1] = c - 1;
158. if (flag == 1)//flag tells that pawn has to be promoted.
159. {
160. printf("\nPromote pawn enter respective numbers:\nqueen\t5\nKnight\t4\nBishop\t3\nRook\t2\n");
161. int p;
162. scanf\_s("%d", &p);
163. Board\_Pieces[y][x] = p;
164. }
165. if ((input[0] >= 0 && input[0] <= 13) && (input[1] >= 0 && input[1] <= 13) && (Moves\_Array[input[1]][input[0]] == 1 || Moves\_Array[input[1]][input[0]] == 3))
166. {
167. if (Board\_Pieces[input[1]][input[0]] == king)//if move results in the killing of a king
168. {
169. int loosing\_side = Player\_Colors[input[1]][input[0]];//color of loosing side is seen and stored
170. num++;
171. lost\_player[num] = loosing\_side;//stored in an array to keep track of all loosing sides
172. //king killed//value of new piece addedd in place of king
173. Player\_Colors[input[1]][input[0]] = Player\_Colors[y][x];
174. Board\_Pieces[input[1]][input[0]] = Board\_Pieces[y][x];
175. Board\_Pieces[y][x] = 0;
176. //board and color of loosing side set to zero to remove pieces of that size
177. for (int y1 = 0; y1 < 14; y1++)
178. {
179. for (int x1 = 0; x1 < 14; x1++)
180. {
181. if (Player\_Colors[y1][x1] == loosing\_side)
182. {
183. Board\_Pieces[y1][x1] = 0;
184. Player\_Colors[y1][x1] = 0;
185. }
186. else
187. continue;
188. }
189. }
190. }
191. else //if piece killed is not the king them simple the killed piece is replaced by the piece that kills it.
192. {
193. Player\_Colors[input[1]][input[0]] = Player\_Colors[y][x];
194. Board\_Pieces[input[1]][input[0]] = Board\_Pieces[y][x];
195. Board\_Pieces[y][x] = 0;
196. }
197. //setting turn of next player
198. switch (Current\_player)
199. {
200. case yellow: (Current\_player = green); break;
201. case green: Current\_player = red; break;
202. case red: Current\_player = blue; break;
203. case blue: Current\_player = yellow; break;
204. }
205. //in case the next player is not in the game anymore
206. for (int a = 1; a < 3; a++)
207. {
208. if (Current\_player != lost\_player[0] && Current\_player != lost\_player[1] && Current\_player != lost\_player[2])
209. break;
210. else
211. Current\_player = (Current\_player == 4) ? 1 : Current\_player + 1;
212. }
213. }
214. //if input is not within range or after killing/moving pieces the moves array is cleared
215. for (int x\_ = 0; x\_ < 14; x\_++)
216. for (int y\_ = 0; y\_ < 14; y\_++)
217. Moves\_Array[x\_][y\_] = 0;
219. flag = 0;
220. }
221. }
222. //function to determine valid piece moves
223. int Valid\_Piece\_Moves(int x, int y)
224. {
225. int Current\_piece = Board\_Pieces[y][x];
226. int Current\_color = Player\_Colors[y][x];
227. int Cant\_move = 1;
228. Moves\_Array[y][x] = 2;
229. if (x > 14 || x < 0 || y > 14 || y < 0 )//in case piece selected is out of range
230. return 1;
231. else if (Current\_piece == 0 || Current\_piece==9)//in case pieces selected is zero or not on board
232. return 1;
233. if (Current\_piece == rook || Current\_piece == queen)//when rook or queen are selected
234. {
235. for (int dir = -1; dir < 2; dir += 2)
236. {
237. int count = 1;
238. while (1) //loop continues until broken
239. {
240. //positive and negative directions along x-axis.
241. if (x + (count \* dir) <= 13 && x + (count \* dir) >= 0)
242. {
243. if (Board\_Pieces[y][x + (count \* dir)] != 0 && Board\_Pieces[y][x + (count \* dir)] != 9)
244. {
245. if (Player\_Colors[y][x + (count \* dir)] != Current\_color)
246. {
247. Moves\_Array[y][x + (count \* dir)] = 3;//moves array set to 3 where piece has the option to capture another piece
248. Cant\_move = 0;
249. }
250. break;
251. }
252. else if (Board\_Pieces[y][x + (count \* dir)] == 0)
253. {
254. Moves\_Array[y][x + (count \* dir)] = 1;// move array set to 1 where piece has valid moves.
255. Cant\_move = 0;
256. }
257. else
258. break;
259. }
260. else
261. break;
262. count++;
263. }
264. count = 1;
265. while (1)
266. {
267. //for positive and negative direction along y-axis.
268. if (y + (count \* dir) <= 13 && y + (count \* dir) >= 0)
269. {
270. if (Board\_Pieces[y + (count \* dir)][x] != 0 && Board\_Pieces[y + (count \* dir)][x] != 9) {
271. if (Player\_Colors[y + (count \* dir)][x] != Current\_color) {
272. Moves\_Array[y + (count \* dir)][x] = 3;
273. Cant\_move = 0;
274. }
275. break;
276. }
277. else if (Board\_Pieces[y + (count \* dir)][x] == 0)
278. {
279. Moves\_Array[y + (count \* dir)][x] = 1;
280. Cant\_move = 0;
281. }
282. else
283. break;
284. }
285. else
286. break;
287. count++;
288. }
289. }
290. }
291. if (Current\_piece == bishop || Current\_piece == queen)//valid moves for bishop and queen
292. {
293. for (int Dir\_X = -1; Dir\_X < 2; Dir\_X += 2) //positive and negative x direction
294. {
295. for (int Dir\_Y = -1; Dir\_Y < 2; Dir\_Y += 2) //poitive and negative y direction
296. {
297. int count = 1;
298. while (1)
299. {
300. if (x + (count \* Dir\_X) <= 13 && x + (count \* Dir\_X) >= 0 && y + (count \* Dir\_Y) <= 13 && y + (count \* Dir\_Y) >= 0)//for moves within range
301. {
302. //for capture of piece
303. if (Board\_Pieces[y + (count \* Dir\_Y)][x + (count \* Dir\_X)] != 0 && Board\_Pieces[y + (count \* Dir\_Y)][x + (count \* Dir\_X)] != 9)
304. {
305. if (Player\_Colors[y + (count \* Dir\_Y)][x + (count \* Dir\_X)] != Current\_color)
306. {
307. Moves\_Array[y + (count \* Dir\_Y)][x + (count \* Dir\_X)] = 3;
308. Cant\_move = 0;
309. }
310. break;
311. }
312. else if (Board\_Pieces[y + (count \* Dir\_Y)][x + (count \* Dir\_X)] == 0)//for valid moves
313. {
314. Moves\_Array[y + (count \* Dir\_Y)][x + (count \* Dir\_X)] = 1;
315. Cant\_move = 0;
316. }
317. else
318. break;
319. }
320. else
321. break;
322. count++;
323. }
324. }
325. }
326. }
327. if (Current\_piece == pawn)//valid moves for pawn
328. {
329. if (Current\_color == red)
330. {
331. if (y - 1 >= 0)
332. {
333. if (Board\_Pieces[y - 1][x] == 0)
334. {
335. flag = (y==1) ? 1 : 0;//to set falg to 1 if pawn reaches second last block
336. Moves\_Array[y - 1][x] = 1;
337. if (y == 12 && Board\_Pieces[y - 2][x] == 0)//for the very first move of pawn
338. Moves\_Array[y - 2][x] = 1;
339. Cant\_move = 0;
340. }
341. if (x + 1 <= 13 && Player\_Colors[y - 1][x + 1] != red && Board\_Pieces[y - 1][x + 1] != 0 && Board\_Pieces[y - 1][x + 1] != 9)//killing piece front right
342. {
343. Moves\_Array[y - 1][x + 1] = 3;
344. Cant\_move = 0;
345. }
346. if (x - 1 >= 0 && Player\_Colors[y - 1][x - 1] != red && Board\_Pieces[y - 1][x - 1] != 0 && Board\_Pieces[y - 1][x - 1] != 9) //killing piece front left
347. {
348. Moves\_Array[y - 1][x - 1] = 3;
349. Cant\_move = 0;
350. }
351. }
352. }
353. else if(Current\_color==yellow)
354. {
355. if (y + 1 <= 13)
356. {
357. if (Board\_Pieces[y + 1][x] == 0)
358. {
359. flag = (y == 12) ? 1 : 0;
360. Moves\_Array[y + 1][x] = 1;
361. if (y == 1 && Board\_Pieces[y + 2][x] == 0)
362. Moves\_Array[y + 2][x] = 1;
363. Cant\_move = 0;
364. }
365. if (x + 1 <= 13 && Player\_Colors[y + 1][x + 1] != yellow && Board\_Pieces[y + 1][x + 1] != 0 && Board\_Pieces[y + 1][x + 1] != 9)
366. {
367. Moves\_Array[y + 1][x + 1] = 3;
368. Cant\_move = 0;
369. }
370. if (x - 1 >= 0 && Player\_Colors[y + 1][x - 1] != yellow && Board\_Pieces[y + 1][x - 1] != 0 && Board\_Pieces[y + 1][x - 1] != 9)
371. {
372. Moves\_Array[y + 1][x - 1] = 3;
373. Cant\_move = 0;
374. }
375. }
376. }
377. else if (Current\_color == green)
378. {
379. if (x - 1 >= 0)
380. {
381. if (Board\_Pieces[y][x-1] == 0)
382. {
383. flag = (x == 1) ? 1 : 0;
384. Moves\_Array[y][x-1] = 1;
385. if (x == 12 && Board\_Pieces[y][x-2] == 0)
386. Moves\_Array[y ][x-2] = 1;
387. Cant\_move = 0;
388. }
389. if (y + 1 <= 13 && Player\_Colors[y + 1][x - 1] != green && Board\_Pieces[y + 1][x - 1] != 0 && Board\_Pieces[y + 1][x - 1] != 9)
390. {
391. Moves\_Array[y + 1][x - 1] = 3;
392. Cant\_move = 0;
393. }
394. if (y - 1 >= 0 && Player\_Colors[y - 1][x - 1] != green && Board\_Pieces[y - 1][x - 1] != 0 && Board\_Pieces[y - 1][x - 1] != 9)
395. {
396. Moves\_Array[y - 1][x - 1] = 3;
397. Cant\_move = 0;
398. }
399. }
400. }
401. else
402. {
403. if (x + 1 <= 13)
404. {
405. if (Board\_Pieces[y][x +1] == 0)
406. {
407. flag = (x == 12) ? 1 : 0;
408. Moves\_Array[y][x + 1] = 1;
409. if (x == 1 && Board\_Pieces[y][x+2] == 0)
410. Moves\_Array[y][x + 2] = 1;
411. Cant\_move = 0;
412. }
413. if (y + 1 <= 13 && Player\_Colors[y + 1][x + 1] != blue && Board\_Pieces[y + 1][x + 1] != 0 && Board\_Pieces[y + 1][x + 1] != 9)
414. {
415. Moves\_Array[y + 1][x + 1] = 3;
416. Cant\_move = 0;
417. }
418. if (y - 1 >= 0 && Player\_Colors[y + 1][x - 1] != blue && Board\_Pieces[y - 1][x + 1] != 0 && Board\_Pieces[y - 1][x + 1] != 9)
419. {
420. Moves\_Array[y + 1][x - 1] = 3;
421. Cant\_move = 0;
422. }
423. }
424. }
425. }
426. if (Current\_piece == knight)//has all eight moves of knight
427. {
428. int horizontal[8] = { 2,1,-1,-2,-2,-1,1,2 };//horizontal translations
429. int vertical[8] = { -1,-2,-2,-1,1,2,2,1 };
430. int current\_row = y;
431. int current\_column = x;
432. for (int move\_numbr = 0; move\_numbr < 8; move\_numbr++)
433. {
434. current\_row += vertical[move\_numbr];//setting row and column according to move postion
435. current\_column += horizontal[move\_numbr];
436. if (current\_row <=13 && current\_column <= 13 && current\_column >= 0 && current\_row >= 0 &&Board\_Pieces[current\_row][current\_column] != 9)
437. {
438. if (Board\_Pieces[current\_row][current\_column] == 0)
439. {
440. Moves\_Array[current\_row][current\_column] = 1;
441. Cant\_move = 0;
442. }
443. else if (Player\_Colors[current\_row][current\_column] != Current\_player)
444. {
445. Moves\_Array[current\_row][current\_column] = 3;
446. Cant\_move = 0;
447. }
448. }
449. current\_row -= vertical[move\_numbr]; //resetting to go back to the initial sqaure of knight to check other moves
450. current\_column -= horizontal[move\_numbr];
451. }
452. }
453. if (Current\_piece == king)
454. {
455. for (int Dir\_X = -1; Dir\_X < 2; Dir\_X++)//positive and negative x directions set.
456. {
457. for (int Dir\_Y = -1; Dir\_Y < 2; Dir\_Y++)//positive and negative y directions set.
458. {
459. if (Dir\_X + x <= 13 && Dir\_X + x >= 0 && y + Dir\_Y >= 0 && Dir\_Y + y <= 14 && !(Dir\_X == 0 && Dir\_Y == 0))
460. {
461. if (Board\_Pieces[Dir\_Y + y][Dir\_X + x] != 0 && Board\_Pieces[Dir\_Y + y][Dir\_X + x] != 9 && Player\_Colors[Dir\_Y + y][Dir\_X + x] != Current\_color)
462. {
463. Moves\_Array[Dir\_Y + y][Dir\_X + x] = 3;
464. Cant\_move = 0;
465. }
466. else if (Board\_Pieces[Dir\_Y + y][Dir\_X + x] == 0) {
467. Moves\_Array[Dir\_Y + y][Dir\_X + x] = 1;
468. Cant\_move = 0;
469. }
470. else
471. break;
472. }
473. }
474. }
475. }
476. return Cant\_move;
477. }
478. void Chess\_Board()
479. {
480. //printing top row
481. printf(" ");
482. for (int y = 0; y < 14; y++) {
483. if (y < 9)
484. printf("%d: ", y + 1);
485. else
486. printf("%d: ", y + 1);
487. }
488. printf("\n\n");
489. for (int x = 0; x < 14; x++)
490. {
491. //printing first column
492. if (x < 9)
493. printf("%d: ", x + 1);
494. else
495. printf("%d: ", x + 1);
496. //printing board
497. for (int y = 0; y < 14; y++)
498. {
499. int piece = Board\_Pieces[x][y];
500. int color = Player\_Colors[x][y];
501. int m = Moves\_Array[x][y];
502. int b = Board\_Pieces[x][y];
503. //for each piece the following steps are followed
504. background\_color(m);//1. background color selected using move array.
505. if (b != 0 && b != 9)
506. {
507. switch (color)
508. {
509. case blue:printf("\x1b[34m%c \x1b[0m", piece\_symbols(piece)); printf("\x1b[0m "); break;
510. case green:printf("\x1b[32m%c \x1b[0m", piece\_symbols(piece)); printf("\x1b[0m "); break;
511. case red:printf("\x1b[31m%c \x1b[0m", piece\_symbols(piece)); printf("\x1b[0m "); break;
512. case yellow:printf("\x1b[33m%c \x1b[0m", piece\_symbols(piece)); printf("\x1b[0m "); break;
513. }
514. }
515. else if (b == 0)
516. {
517. printf("--");
518. printf("\x1b[0m ");
519. }
520. else
521. printf("\x1b[30mxx \x1b[0m");
522. printf(" ");
523. }
524. printf("\n\n");
525. }
526. }
527. void background\_color(int m)
528. {
529. switch (m)
530. {
531. case 3:printf("\x1b[48;5;52m"); break;
532. case 1:printf("\x1b[48;5;22m"); break;
533. case 2:printf("\x1b[48;5;234m"); break;
534. default:printf("\x1b[40m"); break;
535. }
536. }
537. int piece\_symbols(int p)
538. {
539. char c;
540. switch (p)
541. {
542. case pawn:c = 231; break;
543. case rook:c = 254; break;
544. case bishop:c = 30; break;
545. case knight:c = 20; break;
546. case queen:c = 227; break;
547. case king:c = 232; break;
548. }
549. return c;
550. }
551. Output:
552. Screen Displayed on start.
553. Instructions/Rules:
554. Game/board:
555. The following board was randomly set up to demonstrate piece moves.
     1. Player is asked to choose piece(moves are displayed) and then to choose move.
     2. Notice that the players turns change with each move.
556. Pawn moves:
     1. When at initial position: option to move two steps.
     2. When not at initial position: option to move only one step.
     3. Capture option: front right and left of given piece.
     4. Promotion when on the last line.
557. Rook moves: Positive and negative y-axis and x-axis.
558. Bishop moves: all four diagonal directions.
559. Knight moves: L shaped moves.
560. Queen moves: combination of bishop and rook moves.
561. King moves: moves in a circle around the current position.
562. King Killed/End game
     1. When the king of a player is killed, he/she loses the game. (here blue king kills yellow king)
     2. All pieces of the losing side are removed from the board and the game continues.
     3. The last player standing wins the game.