پروژه

شطنرد

ارائه دهنده | عرفان اعتصامی

الگوریتم کلی برنامه

در ابتدا، صفحه‌ی بازی را به ۲ قسمت کلی تقسیم می‌نماییم: ۱)قلمرو بازیکن۱(**C1** و **R1**) و ۲)قلمرو بازیکن۲(**C2** و **R2**) که هر کدام از این بخش ها شامل ۱۲ ستون(هر قلعه ۶ ستون و هر جاده نیز ۶ ستون) هستند. ظرفیت هر ستون را ۶ در نظر گرفته ایم؛ یعنی، در هر ستون حداکثر می‌توان ۶ مهره را قرار داد. به این گونه صفحه‌ی بازی را می‌توان به صورت یک آرایه‌ی سه بعدی در نظر گرفت.

هر بازیکن ۱۵ مهره از ۶ نوع سرباز، فیل، اسب، قلعه، ملکه و شاه دارد. به هر نوع مهره از هر بازیکن، عددی از ۱ تا ۱۲ را نسبت می‌دهیم و خانه‌های آرایه‌ی سه بعدی صفحه‌ی بازی را طبق نقشه‌ی اولیه، مقداردهی می‌کنیم. حال می‌توانیم با انجام عملیات ریاضی مختلف و جابه‌جایی اعداد با یکدیگر، بازی و قوانین آن را پیاده سازی کنیم.

حرکت هر بازیکن در قلمرو خودش به سمت چپ و در قلمرو حریف به سمت راست می‌باشد.

ثابت EMPTY :

مقدار صفر۰ دارد و بیانگر این است که آرایه مورد نظر از خانه‌ها خالی از مهره است. به عنوان مثال آرایه‌ی سه بعدی صفحه‌ی بازی یا آرایه‌ی نشان دهنده‌ی مهره‌های زندانی در ابتدا خالی هستند.

ثوابت مهره ها :

به مهره‌های بازیکن۱، اعداد ۱ تا ۶ را به ترتیب به سرباز، فیل، اسب، قلعه، ملکه و شاه نسبت می‌دهیم و اعداد ۷ تا ۱۲ را به همین ترتیب به مهره‌های بازیکن۲ نسبت می‌دهیم؛ به این ترتیب می‌توانیم با مهره ها مانند اعداد برخورد کنیم.

ثابت ROWS : مقدار ۲ دارد. صفحه‌ی بازی شامل ۲ قسمت کلی است: قلمرو بازیکن۱ و قلمرو بازیکن ۲.

ثابت COLUMNS : مقدار ۱۲ دارد. هر قسمت شامل ۱۲ ستون می‌باشد.

ثابت CAPACITY : مقدار ۶ دارد. هر ستون حداکثر گنجایش ۶ مهره را دارد.

ثابت PIECES : مقدار ۱۵ دارد. هر بازیکن ۱۵ مهره در اختیار دارد.

ثابت TYPES : مقدار ۶ دارد. این ۱۵ مهره شامل ۶ نوع سرباز، فیل، اسب، قلعه، ملکه و شاه هستند.

ثوابت داده‌ها و نتایج درست یا نادرست :

برای ساده‌تر شدن اجرای قوانین بازی، برای حالات و حرکات نادرست، ثوابتی که مقادیر پرتی نسبت به مقادیر ثوابت مربوط به اجزاء اصلی بازی دارند در نظر گرفته ایم. برای حرکت نادرست(**WRONG_MOVE**) که مقدار ۱۰۰- دارد. برای حالتی که حرکتی برای بازیکن باقی نمانده(**NO_MOVES**) که مقدار ۲۰۰- دارد. برای تاسی که استفاده شده(**USED**) که مقدار ۳۰۰- دارد. برای حالت زندانی بودن مهره(**OUT**) که مقدار ۱- دارد. برای حالتی که تابع وظیفه اش را به درستی انجام داده است(**DONE**) که مقدار ۱۰۰ دارد.

ثوابت مربوط به ذخیره‌ی بازی :

برای انجام درست حالاتی که کاربر می‌خواهد بازی را ذخیره کند و شماره گذاری فایل‌های ذخیره شده، نیز ثوابتی را در نظر گرفته ایم. ثابت های **SHOW** و **HIDE** که به ترتیب مقادیر ۱۰۰ و ۱۰۰- را دارند. برای حالتی که ممکن است در روند ذخیره سازی مشکلی پیش آید(**SAVE_INTERRUPTED**) که مقدار ۱۰۰۰- دارد. همچنین برای نام و شماره گذاری فایل ذخیره شده، ثوابت **SavedGame** و **SavedGameNumber** را تعریف کرده ایم که اطلاعات را در قالب فرمت **dat.** ذخیره می‌کنند.

متغیر نام مهره ها(char *names) :

این متغیر که آرایه ای از اشاره‌گرها می‌باشد شامل ۱۳ عنصر است که هر عنصر آن اشاره‌گری به نام مهره ها است که دارای ترتیبی مشابه ثوابت مهره‌ها می‌باشد. به جهت هماهنگ کردن شماره‌ی این عناصر با ثوابت مربوط به آن‌ها یعنی از ۱ تا ۱۲؛ با توجه به این که اندیس یک آرایه از صفر۰ آغاز می‌شود، عنصر نخست این آرایه را به صورت یک رشته که دارای ۴ فاصله(**space**) می‌باشد تعریف کرده‌ایم.

متغیر نام خانه ها(char *rooms) :

این متغیر نیز آرایه‌ای از اشاره‌گرها است که شامل ۲۵ عنصر می‌باشد که هر عنصر آن اشاره‌گری به نام ستون های هر قلمرو از صفحه‌ی بازی است(**C1_1** تا **C1_6**، **R1_1** تا **R1_6**، **C2_1** تا **C2_6** و **R2_1** تا **R2_6**) و عنصر آخر آن **O** می‌باشد که مربوط به بیرون بردن مهره‌ها از بازی برای پیروزی است.

متغیر نام مهره‌های زندانی(char *oNames) :

این متغیر نیز آرایه‌ای از اشاره‌گرها است که شامل ۱۳ عنصر است. هر عنصر آن یک اشاره‌گر به نوع مهره‌ی زندانی شده‌ی هر بازیکن است(**O_S** تا **O_K** برای بازیکن۱ و **O_S** تا **O_K** برای بازیکن۲). جهت هماهنگی شماره‌ی این مهره های زندانی شده با ثوابت مهره‌ها یعنی از ۱ تا ۶ برای بازیکن۱ و از ۷ تا ۱۲ برای بازیکن۲؛ با توجه به این اندیس های آرایه از صفر۰ شروع می‌شوند، عنصر نخست این آرایه را به صورت یک رشته که دارای ۳ فاصله(**space**) می‌باشد تعریف کرده ایم.

آرایه‌ی سه بعدی صفحه‌ی بازی(int board) :

بعد اول این آرایه با توجه به ثابت **ROWS** مقادیر صفر۰(قلمرو بازیکن ۱) و ۱(قلمرو بازیکن۲) را اختیار می‌کند، بعد دوم با توجه به ثابت **COLUMNS** مقادیر صفر۰ تا ۱۱، بعد سوم با توجه به ثابت **CAPACITY** مقادیر صفر۰ تا ۵. این آرایه در ابتدا خالی است و سپس خانه‌های آن با توجه به ورودی های نخستین بازی و جابه‌جایی مهره‌ها، مقادیر عددی ۱ تا ۱۲ را با توجه به ثوابت مهره‌ها به خود می‌گیرند و خانه های خالی آن با مقدار صفر۰ پر می‌شوند.

آرایه‌ی دو بعدی مهره‌های زندانی(int playerOs) :

بعد نخست این آرایه با توجه به این که دو بازیکن داریم مقادیر صفر۰(بازیکن ۱) و ۱(بازیکن ۲) را اختیار می‌کند، بعد دوم با توجه به ثابت **PIECES** مقادیر صفر۰ تا ۱۴(۱۵ مهره). در ابتدا، این آرایه نیز خالی است اما در صورت زندانی شدن مهره‌ای از هر بازیکن، خانه های آن پر می‌شوند.

آرایه‌ی های دو بعدی چاله و سکه(int holes, int coins) :

این دو آرایه ساختاری مشابه با یکدیگر دارند. به این صورت که بعد اول آن ها ۳ مقدار (صفر۰ تا ۲) را به خود می‌گیرد زیرا با توجه به قوانین بازی سه خانه از صفحه‌ی بازی به صورت تصادفی دارای چاله یا سکه می‌شود. بعد دوم که برای آدرس دهی به‌کار می‌رود نیز دارای دو مقدار صفر۰ و ۱ می‌باشد که عنصر صفر۰ بیانگر قلمرویی است که چاله یا سکه در آن قرار دارد و عنصر ۱ بیانگر آن ستون.

آرایه‌ی یک بعدی سکه های جمع شده(int coinsCollected) :

این آرایه دو عنصری همانطور که از نام آن پیدا است، تعداد سکه های جمع شده‌ی توسط هر بازیکن را در خود ذخیره می‌کند. عنصر صفر۰ برای شمارش سکه‌های بازیکن۱ و عنصر ۱ برای سکه‌های بازیکن۲ می‌باشد. در صورتی که بازیکن در خانه‌ای قرار گیرد که شامل سکه است، این سکه به تعداد سکه های آن بازیکن افزوده شده و سپس از صفحه‌ی بازی حذف می‌شود و در صورتی که مهره‌ای از این بازیکن زندانی شود، این بازیکن مختار است که از سکه‌هایش برای آزاد کردن آن مهره‌اش بهره ببرد.

آرایه‌ی یک بعدی امتیاز(int point) :

این آرایه دو عنصری در پایان بازی برای شمارش تعداد مهره‌هایی که هر بازیکن برای پیروزی به بیرون صفحه‌ی بازی منتقل می‌کند به کار می رود؛ به این صورت که، با خارج شدن هر مهره از بازیکن، ۱ واحد به امتیاز آن بازیکن افزوده می‌شود(عنصر صفر۰ برای بازیکن۱ و عنصر۱ برای بازیکن۲). هنگامی که یک بازیکن تمام مهره‌های خودش را خارج کند، امتیاز وی ۱۵ شده و آن بازیکن پیروز می‌شود و بازی به پایان می‌رسد.

آرایه‌ی یک بعدی ممنوعیت حرکت(int playerBanned) :

این آرایه نیز دارای دو عنصر است(عنصر صفر۰ برای بازیکن۱ و عنصر ۱ برای بازیکن۲). در صورتی که مهره‌ی یک بازیکن در چاله بیفتد آن بازیکن برای دو نوبت اجازه‌ی حرکت ندارد. این آرایه برای کنترل ممنوعیت حرکت هر بازیکن به‌کار می‌رود.

متغیر newSaveNumber :

این متغیر در ابتدا مقدار ۱ را دارد و برای شماره‌گذاری فایل های ذخیره شده به ترتیب به‌کار می‌رود.

تابع loadBoard

وظیفه : بارگذاری بازی به سه روش: ۱)فایل‌های ذخیره‌شده ۲)بازی جدید ۳)استفاده از نقشه.

ورودی : ندارد.

خروجی : متغیر صحیح **turn** که بیانگرنوبت است.

```
1. Continue
2. New Game
3. Load From Map File
```

در ابتدا این تابع باعث چاپ یک خروجی به شکل بالا می‌شود که از کاربرد می‌خواهد یکی از ۳ روش گفته شده را انتخاب کند.

حالت بازی جدید :

```
263             board[1][0][0] = K_1;
264             board[1][0][1] = Q_1;
265
266             board[0][0][0] = K_2;
267             board[0][0][1] = Q_2;
268
269             board[0][5][0] = C_1;
270             board[0][5][1] = S_1;
271             board[0][5][2] = H_1;
272             board[0][5][3] = S_1;
273             board[0][5][4] = C_1;
274
275             board[1][5][0] = C_2;
276             board[1][5][1] = S_2;
277             board[1][5][2] = H_2;
278             board[1][5][3] = S_2;
279             board[1][5][4] = C_2;
280
281             board[0][7][0] = S_1;
282             board[0][7][1] = E_1;
283             board[0][7][2] = S_1;
284
285             board[1][7][0] = S_2;
286             board[1][7][1] = E_2;
287             board[1][7][2] = S_2;
288
289             board[1][11][0] = S_1;
290             board[1][11][1] = H_1;
291             board[1][11][2] = S_1;
292             board[1][11][3] = S_1;
293             board[1][11][4] = E_1;
294
295             board[0][11][0] = S_2;
296             board[0][11][1] = H_2;
297             board[0][11][2] = S_2;
298             board[0][11][3] = S_2;
299             board[0][11][4] = E_2;
300
301             createHolesAndCoins();
```

خانه‌های آرایه‌ی سه بعدی صفحه‌ی بازی که شامل مهره هستند، طبق نقشه‌ی اولیه‌ی بازی و مطابق با ثوابت مهره‌ها مقداردهی می‌شوند و سپس چاله‌ها و سکه‌ها در مکان‌های تصادفی از صفحه‌ی بازی قرار می‌گیرند.

حالت فایل‌های ذخیره‌شده :

تابع loadBoard

```
308             FILE *f = fopen(savedGame, "r");
309
310             if(!f) {
311                 printf("\nNo saves found! Try again...");
312                 newSaveNumber = 1;
313                 _getch();
314                 continue;
315             }
316
317             newSaveNumber = loadNames(SHOW);
318
319             do {
320                 printf("\nEnter the number of the save you want: ");
321                 scanf("%d", &saveNumber);
322             } while(saveNumber <= 0 || saveNumber >= newSaveNumber);
```

در ابتدا فایل ذخیره‌ی بازی که ثابت **savedGame** به آن اشاره می‌کند برای خواندن(پرچم **r**) باز می‌شود و در صورت عدم وجود فایل ذخیره، پیام مناسب چاپ می‌شود و در صورت وجود چند فایل از کاربر خواسته می‌شود که شماره‌ی آن فایل را که در لیست نمایان‌شده مشخص است، انتخاب کند.

```
324         for(i = 0 ; i < saveNumber ; i++) {
325             char temp[50];
326
327             fscanf(f, "%s", temp);
328
329             for(m = 0 ; m < ROWS ; m++) {
330                 for(j = 0; j < COLUMNS ; j++) {
331                     for(k = 0 ; k < CAPACITY ; k++) {
332                         fscanf(f, "%d", &board[m][j][k]);
333                     }
334                 }
335             }
336
337             for(m = 0 ; m < 2 ; m++) {
338                 for(j = 0 ; j < PIECES ; j++) {
339                     fscanf(f, "%d", &playerOs[m][j]);
340                 }
341             }
342
343             fscanf(f, "%d%d", &point[0], &point[1]);
344
345             for(m = 0 ; m < 3 ; m++) {
346                 fscanf(f, "%d%d", &holes[m][0], &holes[m][1]);
347             }
348
349             for(m = 0 ; m < 3 ; m++) {
350                 fscanf(f, "%d%d", &coins[m][0], &coins[m][1]);
351             }
352
353             fscanf(f, "%d%d%d%d%d", &playerBanned[0], &playerBanned[1], &coinsCollected[0], &coinsCollected[1], &turn);
354         }
```

سپس صفحه‌ی بازی، مهره‌های زندانی، مهره‌های خارج شده، چاله‌ها، سکه‌ها، ممنوعیت حرکت بازیکن‌ها، سکه‌های جمع‌شده توسط هر بازیکن و نوبت بازی خوانده می‌شود.

حالت استفاده از نقشه :

تابع loadBoard

```
359             int c = 0;
360             char address[100];
361             char temp[5] = {0};
362             int realm = 1;
363             int room = 0;
364             int piece = 0;
365
366             FILE *fMap;
367
368             printf("\n\nMap File Address: ");
369
370             scanf("%s", address);
371
372             fMap = fopen(address, "r");
373
374             if(!fMap) {
375                 printf("Error in openning the map file, Try again!\n");
376                 getch();
377                 continue;
378             }
379
380             while(fgetc(fMap) != '\n');
```

در این حالت، از کاربر خواسته می‌شود که آدرس فایل نقشه را تایپ کند و در صورتی که فایل به درستی باز نشود یا فایل پیدا نشود، پیام مناسب در خروجی چاپ می‌شود.

خط ۳۸۰ : سطر اول فایل نقشه خوانده می‌شود.

```
382             while(!feof(fMap)) {
383                 int i = 0;
384                 int index = 0;
385                 temp[0] = temp[1] = temp[2] = temp[4] = 0;
386                 temp[3] = ' ';
387
388                 if(realm == 0 && piece < 0 && room == 0) {
389                     break;
390                 }
391                 for(i = 0 ; i < 3 ; i++) {
392                     temp[i] = fgetc(fMap);
393
394                     if(temp[i] == '|') {
395                         temp[i] = fgetc(fMap);
396                     }
397                     if(temp[i] == '\n') {
398                         if(realm == 1) {
399                             piece++;
400                         }
401                         else {
402                             piece--;
403                         }
404                         break;
405                     }
406                 }
407                 if(piece >= CAPACITY) {
408                     piece = CAPACITY - 1;
409                     realm = 0;
410                 }
411                 if(temp[i] == '\n') {
412                     board[realm][room][piece] = EMPTY;
413                     room = 0;
414                     temp[0] = temp[1] = temp[2] = temp[3] = temp[4] = 0;
415                 }
416                 else {
417                     for( ; index < COLUMNS+1 && strcmp(temp, names[index]) ; index++);
418                     board[realm][room][piece] = index;
419                     room = (room + 1) % COLUMNS;
420                 }
421             }
422
423             createHolesAndCoins();
```

سپس خطوط موجود در فایل نقشه، کاراکتر به کاراکتر تا هنگامی که به نمایه‌ی پایان فایل برسد، خوانده می‌شوند. سپس در پایان نیز مکان چاله‌ها و سکه‌ها به صورت تصادفی مشخص می‌شود.

خط ۳۹۱ : دلیل این که **i < 3** قراردادهایم، خواندن نام مهره‌ها است که ۳ کاراکتری هستند.

تابع printBoard

وظیفه : چاپ صفحه‌ی بازی به همراه جدول مهره‌های زندانی

ورودی : ندارد.

خروجی : ندارد.

```
439 void printBoard(void) {
440
441     int outCounter = 0;
442     int i, j;
443
444     system("cls");
445
446     printf("Player2's Castle\t\tRoad number 2\t\t\tPlayer1 Cell | Player2 Cell\n");
447     printf(" 1   2   3   4   5   6  |");
448     printf(" 1   2   3   4   5   6\t\t\t\t     |\n");
449
450     for(i = 0 ; i < CAPACITY ; i++) {
451         for(j = 0 ; j < COLUMNS ; j++) {
452             printf("%s", names[board[1][j][i]]);
453             if(j == 5) {
454                 printf("|");
455             }
456         }
457         printf("\t\t\t%s  |  %s\n", oNames[playerOs[0][outCounter]], oNames[playerOs[1][outCounter]]);
458         outCounter++;
459     }
460
461     for(i = CAPACITY-1 ; i >= 0 ; i--) {
462         for(j = 0 ; j < COLUMNS ; j++) {
463             printf("%s", names[board[0][j][i]]);
464             if(j == 5) {
465                 printf("|");
466             }
467         }
468         printf("\t\t\t%s  |  %s\n", oNames[playerOs[0][outCounter]], oNames[playerOs[1][outCounter]]);
469         outCounter++;
470     }
471
472     printf(" 1   2   3   4   5   6  |");
473     printf(" 1   2   3   4   5   6   ");
474     printf("\t\t\t%s  |  %s\n", oNames[playerOs[0][outCounter]], oNames[playerOs[1][outCounter]]);
475     printf("Player1's Castle\t\tRoad number 1\n");
476
477     return;
478 }
```

متغیر **outCounter** تعریف شده در ابتدای تابع برای چاپ کردن به ترتیب مهره‌های زندانی به صورت زیر هم به‌کار می‌رود. فرآیند چاپ صفحه‌ی بازی به دو جفت حلقه‌ی تودرتو تقسیم شده است که جفت اول، قسمت بالای صفحه را(قلمرو بازیکن ۲) و جفت دوم قسمت پایین صفحه را(قلمرو بازیکن ۱) چاپ می‌کند.

خط ۴۴۴ : برای جلوگیری از شلوغ شدن محیط اجرای بازی و عدم نیاز به اسکرول، هرگاه این تابع فراخوانی شود، در ابتدای کار محیط اجرای بازی توسط این دستور پاک می‌شود.

تابع findFreePiece

وظیفه : یافتن روترین مهره‌ی قابل حرکت

ورودی : متغیر صحیح **realm**(قلمرو) و متغیر صحیح **room**(ستون).

خروجی : متغیر صحیح **x** که بیانگر جای روترین مهره‌ی قابل حرکت در قلمرو و ستون مورد نظر است.

```
872  //find which piece of the specific part and column has to move
873  //realm : free piece's part
874  int findFreePiece(int realm, int room) {
875
876      int x = CAPACITY - 1;
877
878      for(x ; x >= 0 && board[realm][room][x] == EMPTY ; x--);
879
880      return x; //returing index of the row of that specific realm(part) and room(column)
881  }
```

چون اندیس های آرایه از صفر۰ آغاز می‌شوند، در ابتدا **x** را برابر با **CAPACITY-1** قرار دادیم. حلقه‌ی داخل تابع روترین مهره را مکان‌یابی کرده و بازمی‌گرداند.

تابع createHolesAndCoins

وظیفه : ایجاد چاله و سکه در مکان‌های تصادفی

ورودی : ندارد.

خروجی : ندارد.

```
473 void createHolesAndCoins(void) {
474
475     int num = rand() % 3;
476     int i;
477
478     for(i = 0 ; i <= num ; i++) {
479         do {
480             holes[i][0] = rand() % ROWS;
481             holes[i][1] = rand() % (COLUMNS/2) + (COLUMNS / 2);
482         } while(board[holes[i][0]][holes[i][1]][0] != EMPTY);
483     }
484
485     num = rand() % 3;
486
487     for(i = 0 ; i <= num ; i++) {
488         do {
489             coins[i][0] = rand() % ROWS;
490             coins[i][1] = rand() % COLUMNS;
491         } while(board[coins[i][0]][coins[i][1]][0] != EMPTY || (coins[i][0] == holes[i][0] && coins[i][1] == holes[i][1]));
492     }
493
494     return;
495 }
```

در ابتدا عدد رندومی بین صفر ۰ تا ۲ تولید شده و در متغیر **num** ذخیره می‌شود. سپس حلقه‌ی **for** اول در تابع اجرا شده که حلقه **do_while** داخل آن وظیفه‌ی آدرسی دهی تصادفی به مکان چاله ها را دارد. سپس همین روند برای تولید سکه‌ها نیز به‌کار می‌رود.

خط ۴۸۱ : دلیل افزودن **COLUMNS/2** این است که در قوانین گفته شده که چاله‌ها در قسمت‌های جاده‌ی صفحه‌ی بازی قرار می‌گیرند.

خط ۴۹۱ : قسمت دوم شرط داخل پرانتزهای **while** به دلیل هم‌نهشتی احتمالی جای چاله‌ها و سکه‌هاست.

تابع checkForCoinRelease

وظیفه : آزادسازی اختیاری مهره‌ی زندانی شده و جاگذاری آن در یکی از خانه‌های قلعه‌ی بازیکن حریف.

ورودی : متغیر صحیح **player**(صفر برای بازیکن ۱ و ۱ برای بازیکن ۲) و متغیر **cellFirstFree** که بیانگر روترین مهره‌ی زندانی شده در جدول مهره‌های زندانی است.

خروجی : ندارد.

```
847 void checkForCoinRelease(int player, int cellFirstFree) {
848
849     if(coinsCollected[player] > 0) {
850         int search;
851
852         //check for first empty room
853         for(search = 0 ; search < COLUMNS/2 ; search++) {
854             if(board[(player + 1) % 2][search][0] == EMPTY) {
855                 break;
856             }
857         }
858
859         if(search < COLUMNS/2) {
860             char c;
861
862             printf("\n\nPlayer %d, do you want to use your coin? y. Yes n. No : ", player+1);
863
864             c = _getch();
865
866             if(c == 'y' || c == 'Y') {
867                 coinsCollected[player]--;
868                 //moving out piece to particular room of the oppsite player's castle
869                 board[(player + 1) % 2][search][0] = playerOs[player][cellFirstFree];
870                 //removing the out piece from cell
871                 playerOs[player][cellFirstFree] = EMPTY;
872             }
873         }
874     }
875
876     return;
877 }
```

در ابتدای کار بررسی می‌شود که آیا اصلاً بازیکن مشخص شده، سکه‌ای برای استفاده دارد یا خیر. سپس در صورت مثبت بودن جواب، حلقه موجود با استفاده از متغیر **search** به دنبال نخستین ستون خالی در قلعه بازیکن حریف می‌گردد. در صورت یافت شدن این ستون، از بازیکن پرسیده می‌شود که آیا قصد استفاده از سکه‌ی خود را دارد یا خیر. در صورت مثبت بودن پاسخ، اولین مهره‌ی موجود در زندان به خانه‌ی پیدا شده در صفحه‌ی بازی منتقل می‌شود.

تابع freeThePiece

وظیفه : آزادکردن مهره‌ها از زندان و قرار دادن آن‌ها در جای مناسب از خانه‌های قلعه‌ی حریف.

ورودی : متغیر صحیح **player**(صفر ۰ برای بازیکن۱ و ۱ برای بازیکن۲)، متغیر صحیح **pIndex** که بیانگر اندیس مهره است و متغیر صحیح **dice** که بیانگر مقدار تاس است.

خروجی : مقدار **WRONG_MOVE** برای نادرستی حرکت و مقدار **DONE** در صورت انجام شدن درست فرآیند.

فقط به شرح قسمت‌هایی که شاید کمی مبهم باشند، می‌پردازیم.

```
925     for(i = pIndex ; i < PIECES-1 ; i++) {
926         playerOs[player][i] = playerOs[player][i + 1];
927     }
```

این حلقه برای این است که همواره مطمئن باشیم، اولین مهره‌ی زندانی در عنصر صفر۰ام زندان قرار دارد.

تابع obtainMove

وظیفه : تشخیص مبدأ و مقصد حرکت.

ورودی : آرایه یک بعدی **s** که بیانگر آدرس مبدأ ورودی کاربر است، آرایه یک بعدی **d** که بیانگر آدرس مقصد ورودی کاربر است و آرایه یک بعدی **r** که آدرس مبدأ و مقصد را در خود ذخیره می‌کند.

خروجی : ندارد.

منظور از آدرس، قلمرو و ستون است.

```
604  //s : source - d : desination - r : changing moveInfo elemnts
605  void obtainMove(char *s, char *d, int *r) {
606
607      int i;
608
609      r[0] = 0; //to check whether both source and destination(or each of them) are correct or not
610
611      for(i = 0 ; i < COLUMNS*ROWS+1 ; i++) {
612          //validating and finding the source
613          if(!strcmp(s, rooms[i])) {
614              r[1] = i / 12; //which part
615              r[2] = i % 12; //which column(room)
616              r[0]++;
617          }
618          //validating and findong the destination
619          if(!strcmp(d, rooms[i])) {
620              r[3] = i / 12; //which part
621              r[4] = i % 12; //which column(room)
622              r[0]++;
623          }
624      }
625
626      return;
627  }
```

حلقه‌ی **for**، وظیفه‌ی مطابقت دادن ستون مبدأ و مقصد وارده را با ستون‌های پیش‌فرض بازی دارد.

خط ۶۰۹ : عنصر صفر•ام این آرایه برای مشخص کردن درستی یا نادرستی مبدأ و مقصد وارده به‌کار می‌رود به گونه‌ای که اگر هر دو درست باشند مقدار ۲، فقط یک‌کدام، مقدار ۱ و اگر هیچ‌کدام درست نباشند، مقدار صفر• را دارد.

تابع selectDice

وظیفه : جابه‌جایی پذیری تاس‌ها و مطابقت تاس‌ها با حرکت.

ورودی : متغیر صحیح **player**(صفر۰ برای بازیکن۱ و ۱ برای بازیکن۲)، متغیرهای صحیح **sRealm**، **sRoom**، **dRealm**، **dRoom** که به ترتیب نمایانگر قلمرو مبدأ، ستون مبدأ، قلمرو مقصد، ستون مقصد هستند، آرایه یک بعدی **dices** که حاوی مقدار تاس‌هاست و اشاره‌گر **matched** که قرار است آن را در صورت مطابقت یا عدم مطابقت بازگردانی کنیم.

خروجی : مقدار صفر۰ برای عنصر نخست آرایه‌ی تاس‌ها و مقدار ۱ برای عنصر دوم آرایه‌ی تاس‌ها.

فقط به شرح قسمت‌هایی که شاید کمی مبهم باشند، می‌پردازیم.

```
657     //dice[0] was used
658     if(dices[0] == USED) {
659         return 1;
660     }
661     //dice[1] was used
662     if(dices[1] == USED) {
663         return 0;
664     }
```

این بخش‌ها بررسی می‌کنند که کدام تاس قبلاً استفاده شده‌است و اندیس عنصر تاس دیگر را برمی‌گردانند.

```
631                                                 //the movement is leftward when we don't have change in realm and is rightward when we have change in realm
632 int step = player == sRealm ? -1 : 1; //show heading -- if step == -1 : decreases in index(leftward)  , step == 1 : increases in index(rightward)
```

متغیر **step** بیانگر جهت حرکت مهره‌ها در قلمرویی است که در آن هستند.

```
677     else if(dRealm != 2) { //whether dRealm is 0 or 1 -- if dRealm == 2 -> it means the destination is O(victory process)
678         if(step != -1) { //the move should be leftward
679             distance = (COLUMNS - sRoom) + (COLUMNS - dRoom) - 1;
680
681             if(distance == dices[0]) {
682                 *matched = 1; //the movement is corresponding with dices
683                 return 0;
684             }
685
686             if(distance == dices[1]) {
687                 *matched = 1; //the movement is corresponding with dices
688                 return 1;
689             }
690         }
691     }
```

در صورتی که **dRealm != 2** باشد، یعنی حرکت به جهت خارج کردن مهره نباشد، فاصله‌ی مبدأ با مقصد سنجیده شده و با تاس ها مطابقت داده می‌شود.

تابع anyMoveLeft

وظیفه : بررسی حرکات باقی‌مانده.

ورودی : متغیر صحیح **player**(صفر ۰ برای بازیکن۱ و ۱ برای بازیکن۲) و متغیر صحیح **dice** که بیانگر مقدار تاس است.

خروجی : مقدار ۱ برای حالتی که بازیکن می‌تواند حرکت کند و مقدار صفر ۰ برای حالتی که نمی‌تواند.

این تابع به دو بخش کلی مشابه یکدیگر تقسیم می‌شود. هر بخش مخصوص هر بازیکن است که یکی از آن‌ها را مورد بررسی قرار می‌دهیم.

```
512         //check if there is any piece in cell and check if player can take piece inside the board with that dice
513         if(playerOs[0][0] != EMPTY) {
514             if(board[1][dice - 1][1] <= K_1 && board[1][dice - 1][CAPACITY - 1] != EMPTY) {
515                 return 0;
516             }
517             else if(board[1][dice - 1][1] >= S_2) {
518                 return 0;
519             }
520         }
```

این بخش بررسی می‌کند که آیا بازیکن، مهره‌ای در زندان دارد یا خیر. در صورت مثبت بودن پاسخ، باید این مهره‌ی زندانی در صورت وجود فضا به قلعه‌ی بازیکن حریف منتقل شود، اگر فضایی برای انتقال آن مهره نبود، بازیکن تا مهره‌ای در زندان دارد نمی‌تواند از تاس های خود برای حرکت سایر مهره‌ها استفاده کند.

```
522         if(board[0][i][0] <= K_1 && board[0][i][0] > EMPTY) {
523             r1 = (i + 1 - dice); //player1 move to left in its realm -- destination room
524
525             if(r1 == 0) { //can move out of the board
526                 return 1;
527             }
528
529             if(r1 > 0) { //can move but with conditions
530                 if(board[0][i - dice][1] <= K_1 && board[0][i - dice][CAPACITY - 1] == EMPTY) {
531                     return 1;
532                 }
533             }
534         }
```

این بخش به حرکت مهره‌های بازیکن در قلمرو خودش می‌پردازد. اگر مهره‌ی بازیکن یافت شد می‌تواند در صورت باقی‌بودن بقیه‌ی شرایط(پر بودن قلعه) آن را به بیرون منتقل کند یا آن را اگر قوانین بازی(وجود مهره‌های خودش در خانه‌ی مقصد یا خانه‌ی خالی مقصد و ...) اجازه دهند به خانه‌ای دیگر منتقل کند.

```
536         if(board[1][i][0] <= K_1 && board[1][i][0] > EMPTY) {
537             //player1 move to right in player2's realm -- destination room
538             r1 = (i + dice) % COLUMNS;
539             r2 = (i + dice) / COLUMNS;
540
541             if(r2 == 0) { //if i+dice < 12 -- this move doesn't change the realm
542                 if(board[1][i + dice][1] <= K_1 && board[1][i + dice][CAPACITY - 1] == EMPTY) {
543                     return 1;
544                 }
545             }
546             else { //if i+dice > 12 -- this move changes the realm
547                 if(board[0][COLUMNS - r1 - 1][1] <= K_1 && board[0][COLUMNS - r1 - 1][CAPACITY - 1] == EMPTY) {
548                     return 1;
549                 }
550             }
551         }
```

این بخش هم مشابه بالا به حرکت بازیکن در قلمرو بازیکن حریف می‌پردازد.

تابع anyMoveLeft

```
597     return isPlayerCastleFull(player) && ((dice >= 5 && ((player == 0 && board[player][4][0] <= K_1 && board[player][4][0] > EMPTY) || (player == 1 && board[player][4][0] >= S_2))) ||
598                                          (dice >= 4 && ((player == 0 && board[player][3][0] <= K_1 && board[player][3][0] > EMPTY) || (player == 1 && board[player][3][0] >= S_2))) ||
599                                          (dice >= 3 && ((player == 0 && board[player][2][0] <= K_1 && board[player][2][0] > EMPTY) || (player == 1 && board[player][2][0] >= S_2))) ||
600                                          (dice >= 2 && ((player == 0 && board[player][1][0] <= K_1 && board[player][1][0] > EMPTY) || (player == 1 && board[player][1][0] >= S_2))) ||
601                                          (dice >= 1 && ((player == 0 && board[player][0][0] <= K_1 && board[player][0][0] > EMPTY) || (player == 1 && board[player][0][0] >= S_2))) );
602
```

هدف از این **return**، اجرای قانون ۱۰ بازی است.

تابع movePlayer

وظیفه : حرکت دادن مهره‌ها.

ورودی : متغیر صحیح **player**(صفر ۰ برای بازیکن ۱ و ۱ برای بازیکن ۲)، متغیر صحیح **dice** که بیانگر عدد تاس می‌باشد. متغیرهای صحیح **sRealm**، **sRoom**، **dRealm**، **dRoom** که به ترتیب نمایانگر قلمرو مبدأ، ستون مبدأ، قلمرو مقصد، ستون مقصد هستند.

خروجی : مقدار **WRONG_MOVE** برای نادرستی حرکت و مقدار **DONE** در صورت انجام شدن درست فرآیند.

فقط به شرح قسمت‌هایی که شاید کمی مبهم باشند، می‌پردازیم.

```
744     int nextPlayer = (player + 1) % 2;
```

متغیر **nextPlayer** بیانگر اندیس بازیکن حریف است.

```
758     if(board[dRealm][dRoom][0] == EMPTY) {
759         dFirstFree = 0;
760     }
761     else {
762         dFirstFree = findFreePiece(dRealm, dRoom) + 1;
763     }
```

در صورتی که ستون مقصد خالی باشد، مهره باید ردیف اول آن قرار گیرد و در صورتی که حاوی مهره باشد، باید روی روترین ردیف پس از مهره‌ها قرار گیرد.

تابع isPlayerCastleFull

وظیفه : مشخص کردن وضعیت پربودن قلعه‌ی بازیکن(جمع شدن همه‌ی مهره‌های آن بازیکن در خانه‌های قلعه‌اش).

ورودی : متغیر صحیح **player**(صفر ۰ برای بازیکن۱ و ۱ برای بازیکن۲).

خروجی : متغیر صحیح **flag** که بیانگر درستی یا نادرستی پربودن قلعه‌ی بازیکن است.

```
945 int isPlayerCastleFull(int player) {
946
947     int numberOfPiecesInCastle = 0;
948     int flag = 0;
949     int i, j;
950
951     for(i = 0 ; i < COLUMNS/2 ; i++) {
952         for(j = 0 ; j < CAPACITY && board[player][i][j] != EMPTY ; j++) {
953             if(player == 0 && board[player][i][j] > EMPTY && board[player][i][j] <= K_1) {
954                 numberOfPiecesInCastle++;
955             }
956             else if(player == 1 && board[player][i][j] >= S_2) {
957                 numberOfPiecesInCastle++;
958             }
959         }
960     }
961
962     if(numberOfPiecesInCastle == PIECES - point[player]) {
963         flag = 1;
964     }
965
966     return flag;
967 }
```

توسط حلقه‌های تودرتوی داخل تابع، به دنبال مهره‌های هر بازیکن در خانه های قلعه‌اش می‌گردیم و با هر بار یافت شدن آن‌ها، یک واحد به متغیر **numberOfPiecesInCastle** افزوده می‌شود. سپس در پایان بررسی می‌کنیم که آیا این تعداد برابر با تعداد کل مهره‌ها منهای مهره‌هایی که تاکنون خارج شده‌اند، هست یا خیر. در صورت مثبت بودن پاسخ متغیر **flag** با مقدار ۱ بازگردانی می‌شود و در غیر این صورت با مقدار صفر ۰.

تابع movePieceToO

وظیفه : انتقال مهره‌ها به خارج از بازی در فرآیند پیروزی.

ورودی : متغیر صحیح **player**(صفر ۰ برای بازیکن۱ و ۱ برای بازیکن۲)، متغیر صحیح **sRealm** که بیانگر قلمرو مبدأ می‌باشد، متغیر صحیح **sRoom** که بیانگر ستون مبدأ است و متغیر صحیح **dice** که مقدار تاس را در خود دارد.

خروجی : مقدار **WRONG_MOVE** برای نادرستی حرکت و مقدار **DONE** در صورت انجام شدن درست فرآیند.

```
969  //victory process
970  //sRealm : source's part , sRoom : source's room
971  int movePieceToO(int player, int sRealm, int sRoom, int dice) {
972
973      int flag = 0;
974      int piece = findFreePiece(sRealm, sRoom);
975      int numberOfPiecesInCastle = 0;
976      //check whether that specifc place is empty or not
977      if(board[sRealm][sRoom][0] == EMPTY) {
978          return WRONG_MOVE;
979      }
980      //check that each player moves out her/his own pieces -- player == 0 -> player1 -- player == 1 -> player2
981      if((player == 0 && board[sRealm][sRoom][0] >= S_2) || (player == 1 && board[sRealm][sRoom][0] <= K_1)) {
982          return WRONG_MOVE;
983      }
984
985      flag = isPlayerCastleFull(player);
986
987      if((sRealm == player && sRoom == dice-1) || (flag == 1 && ((sRoom == 4 && dice >= 5) || (sRoom == 3 && dice >= 4) ||
988                                                                 (sRoom == 2 && dice >= 3) || (sRoom == 1 && dice >= 2) ||
989                                                                 (sRoom == 0 && dice >= 1)))) {
990          //number of outed piece
991          point[player]++;
992          //removing the piece from the board
993          board[sRealm][sRoom][piece] = EMPTY;
994          //checking the process of reaching victory
995          if(point[player] == PIECES) {
996              printf("\n*! Player %d Won !* ", player+1);
997              _getch();
998              exit(1);
999          }
1000     }
1001     else {
1002         return WRONG_MOVE;
1003     }
1004
1005     return DONE;
1006 }
```

ابتدا مشخص می‌شود که مبدأ خالی از مهره نباشد و همچنین هر بازیکن مهره‌ی خود را حرکت دهد. سپس پربودن یا نبودن قلعه را مشخص می‌کنیم. سپس در صورت درست بودن قلمرو و مطابقت ستون با تاس، مهره ای بازی خارج شده و یک واحد به امتیاز بازیکن افزوده می‌شود. در صورتی که بازیکن، همه‌ی مهره‌های خود را خارج کند، پیروز بازی شده و بازی به پایان می‌رسد.

خطوط ۹۸۷ تا ۹۸۹ : شرط چند قسمتی داده شده برای اجرای قانون شماره ۱۰ بازی است.

تابع showMsg

وظیفه : چاپ پیام خروجی مناسب

ورودی : متغیر صحیح **player**(صفر، برای بازیکن۱ و ۱ برای بازیکن۲)، متغیر صحیح **r**(یک نوع **response**)، متغیر صحیح **i** و **moves** که برای مقایسه‌ی درستی تعداد دفعات حرکت بازیکن هستند.

خروجی : ندارد.

```
705  //show appropriate prompt
706  void showMsg(int player, int r, int i, int moves) {
707
708      char *moveOrders[4] = {"first", "second", "third", "forth"};
709
710      if(r == WRONG_MOVE) {
711          printf("\nWrong move Player %d, please try again: ", player);
712          return;
713      }
714      else if(r == NO_MOVES) {
715          printf("\nplayer %d, No moves available", player);
716          return;
717      }
718      else if(r == 0) {
719          printf("\nPlayer %d, please enter your %s source and destination: ", player, moveOrders[i]);
720          return;
721      }
722      else if(i == moves) {
723          printf("\nNice move Player %d, end of your turn...", player);
724          return;
725      }
726      else {
727          printf("\nNice move Player %d , now please enter your %s source and destination: ", player, moveOrders[i]);
728          return;
729      }
730
731  }
```

خط ۷۰۸ : دلیل ایجاد این آرایه از اشاره‌گر ها، کاربرپسندتر کردن بازی و مشخص کردن این است که بازیکن در حال انجام چندمین حرکت خود است.

تابع loadNames

وظیفه : شماره‌گذاری فایل های ذخیره‌شده به ترتیب و بازگردانی شماره‌ی جدید.

ورودی : متغیر صحیح **readMode** که بیانگر حالتی است که بخواهیم فایل‌های ذخیره‌شده را نشان دهیم.

خروجی : متغیر صحیح **newSaveNumber** که شماره جدید فایل‌های ذخیره شده است.

```
1008    //returning number of the saved games
1009    int loadNames(int readMode) {
1010
1011        int i;
1012
1013        FILE *fSavedNames = fopen(savedGameNumber, "r");
1014
1015        newSaveNumber = 1;
1016
1017        if(!fSavedNames) {
1018            return 1;
1019        }
1020
1021        fscanf(fSavedNames, "%d", &newSaveNumber);
1022
1023        newSaveNumber++;
1024
1025        if(readMode == SHOW) {
1026            printf("\n\nSaves:\n");
1027            for(i = 1 ; i < newSaveNumber ; i++) {
1028                printf("%d. Shatnard%d\n", i, i);
1029            }
1030        }
1031
1032        return newSaveNumber;
1033    }
```

ابتدا فایل **dat.** ذخیره شده(**numberOfSavedGames**) برای خواندن(پرچم **r**) باز می‌شود. سپس شماره جدید از فایل **fSavedNames** خوانده می‌شود و یک واحد به آن افزوده می‌شود. سپس در صورتی که کاربر قصد ادامه‌ی بازی از فایل‌های ذخیره‌شده را داشته باشد، متغیر **readMode** برابر ثابت **SHOW** شده و نام فایل‌های ذخیره شده نشان داده می‌شود.

تابع save

وظیفه : ذخیره سازی.

ورودی : متغیر صحیح **turn** که بیانگر شماره نوبت بازیکنی است که در حال حرکت است.

خروجی : مقدار **SAVE_INTERRUPTED** برای نادرستی فرآیند ذخیره‌سازی و مقدار **DONE** برای درستی.

```
1035  int save(int turn) {
1036
1037      int i, j, k;
1038
1039      FILE *f = fopen(savedGame, "a");
1040      FILE *fSaveNames = fopen(savedGameNumber, "w");
1041
1042      if(!f) {
1043          f = fopen(savedGame, "w");
1044
1045          newSaveNumber = 1;
1046
1047          if(!f || !fSaveNames) {
1048              return SAVE_INTERRUPTED;
1049          }
1050      }
1051
1052      fprintf(f, "Shatnard%d\n", newSaveNumber);
1053      fprintf(fSaveNames, "%d\n", newSaveNumber);
1054
1055      fclose(fSaveNames);
```

در ابتدا فایل ذخیره‌ی بازی برای نوشتن در انتهای آن(پرچم **a**) و سپس فایل حاوی شماره‌ی ذخیره‌سازی برای نوشتن(پرچم **w**) گشوده می‌شوند. در صورتی که اولین بار باشد که بازی قرار است ذخیره شود، فایل ذخیره‌ی بازی(**gameSave**) برای نوشتن(پرچم **w**) باز می‌شود. سپس نام ذخیره‌سازی با فرمت گفته شده در قوانین در فایل‌ها نوشته می‌شود.

```
1056      //saving board
1057      for(i = 0 ; i < ROWS ; i++) {
1058          for(j = 0 ; j < COLUMNS ; j++) {
1059              for(k = 0 ; k < CAPACITY ; k++) {
1060                  fprintf(f, "%d\n", board[i][j][k]);
1061              }
1062          }
1063      }
1064      //saving out pieces
1065      for (i = 0 ; i < 2 ; i++) {
1066          for(j = 0 ; j < PIECES ; j++) {
1067              fprintf(f, "%d\n", playerOs[i][j]);
1068          }
1069      }
1070      //saving points gained in victory process
1071      fprintf(f, "%d\n%d\n", point[0], point[1]);
1072      //saving holes
1073      for(i = 0 ; i < 3 ; i++) {
1074          fprintf(f, "%d\n%d\n", holes[i][0], holes[i][1]);
1075      }
1076      //saving coins
1077      for(i = 0 ; i < 3 ; i++) {
1078          fprintf(f, "%d\n%d\n", coins[i][0], coins[i][1]);
1079      }
1080      //saving banned players and collected coins
1081      fprintf(f, "%d\n%d\n%d\n%d\n%d\n", playerBanned[0], playerBanned[1], coinsCollected[0], coinsCollected[1], turn);
1082
1083      fclose(f);
1084
1085      newSaveNumber++;
1086
1087      return DONE;
1088  }
```

تابع save

برای ذخیره‌ی بازی نیز اطلاعات مورد نیاز شامل موارد زیر به ترتیب در فایل ذخیره‌ی بازی بارگذاری می‌شوند: صفحه‌ی بازی، مهره‌های زندانی، مهره‌های خارج شده، چاله ها، سکه ها، ممنوعیت بازیکن ها از حرکت، سکه های جمع شده توسط هر بازیکن و درنهایت نوبت بازی. سپس فایل بسته شده و مقدار **DONE** بازگردانده می‌شود.

برخی نکات مهم :

- با توجه به داده‌های ورودی، پیام مناسب چاپ می‌شود.
- پس از تمام شدن حرکات یک بازیکن، پرسیده می‌شود که آیا کاربر مایل است بازی را ذخیره‌کند یا به سراغ ادامه‌ی بازی رود.
- وجود توابع **getch()** در بسیاری از قسمت‌ها، برای این است که زمان کافی برای نمایش پیام ها وجود داشته‌باشد.
- در صورت نادرست بودن یک ورودی، دوباره از کاربر خواسته می‌شود که ورودی خود را تایپ کند.