Programming Assignment #1: ChessMoves

COP 3502, Fall 2016

Due: Sunday, September 25, *before* 11:59 PM

Table of Contents

Abstract

In this programming assignment, you will implement an interpreter for algebraic chess notation, which is a system used to describe the moves made over the course of a chess game. In order to finish this program, you will need to learn a bit about chess: how pieces can move on the game board, as well as how those moves are denoted with algebraic chess notation.

By completing this assignment, you will gain experience manipulating strings, 2D arrays, structs, and struct pointers in C. You will also gain experience working with dynamic memory allocation and squashing memory leaks. On the software engineering side of things, you will learn to construct programs that use multiple source files and custom header (.h) files. This assignment will also help you hone your ability to read technical specifications and implement a reasonably lengthy, non-trivial project with interconnected pieces.

Attachments

ChessMoves.h
testcase{01-20}.c
output{01-20}.txt
test-all.sh

Deliverables

ChessMoves.c

(***Note!*** Capitalization and spelling of your filename matters!)

1. Overview

There are a few different notation systems used to denote the moves made in a game of chess. One of those systems is called “algebraic chess notation.” With algebraic chess notation, a sequence of moves made in a chess game is recorded as a series of strings that look something like this:[1]

1. e4 e5
2. Nf3 Nc6
3. Bb5 a6
4. Ba4 Nf6
5. 0-0 Be7

For this assignment, you will write a program that parses through a series of algebraic chess notation strings and prints out what the chess board looks like with each move that is made. To do this, you will first have to learn (or brush up on) how chess pieces move (see Appendix A in this document) and how algebraic chess notation works (see Appendix B).

A complete list of the functions you must implement, including their functional prototypes, is given below in Section 5, “Function Requirements”. You will submit a single source file, named `ChessMoves.c`, that contains all required function definitions, as well as any auxiliary functions you deem necessary. In `ChessMoves.c`, you will have to #include any header files necessary for your functions to work, including the custom `ChessMoves.h` file we have distributed with this assignment (see Section 4, “ChessMoves.h”).

Note: You will *not* write a main() function in the source file you submit! Rather, we will compile your source file with our own `main()` function(s) in order to test your code. We have attached example source files that have `main()` functions, which you can use to test your code. You can write your own `main()` functions for testing purposes, but the code you submit must not have a `main()` function. We realize this is completely new territory for most of you, so don’t panic. We’ve included instructions for compiling multiple source files into a single executable (e.g., mixing your `ChessMoves.c` with our `ChessMoves.h` and `testcaseXX.c` files) in Sections 6 and 7 of this PDF.

Although we have included test cases with sample `main()` functions to get you started with testing the functionality of your code, we encourage you to develop your own test cases, as well. Ours are by no means comprehensive. We will use much more elaborate test cases when grading your submission.

Start early. Work hard. Good luck!

2. Important Note: Test Case Files Look Wonky in Notepad

Included with this assignment are several test cases, along with output files showing exactly what your output should look like when you run those test cases. You will have to refer to those as the gold standard for how your output should be formatted. Please note that if you open those files in Notepad, they will appear to be one long line of text. That’s because Notepad handles end-of-line characters differently from Linux and Unix-based systems. One solution is to view those files in an IDE (like CodeBlocks), or you could use a different text editor (like Atom or Sublime).

1 This sequence comes from an actual game played by chess grandmasters Kasparov and Karpov. (*source*)

3. Adopting a Growth Mindset

A word of advice before we dive in to the details: When faced with a large, challenging assignment like this, it's important not to look at it as an instrument that is being used to measure how much you already know (whether that's knowledge of programming, chess, or math, or even what some might call "natural intellectual capability"). Rather, it's important to view this assignment as a chance to learn something new, grow your skill set, and embrace a difficult challenge. It's also important to view intellectual capability as something that can grow and change over one's lifetime. Adopting that mindset will allow you to reap greater benefits from this assignment (and from all your college-level coursework) regardless of the grades you earn.

For more on the importance of adopting a growth mindset throughout your academic career and how that can impact your life, see the following:

> Growth Mindset vs Fixed Mindset: An Introduction (Watch time: 2 min 42 sec)
>
> The Power of Belief – Mindset and Success (Watch time: 10 min 20 sec)

4. ChessMoves.h

Included with this assignment is a customer header file that includes the struct definitions and functional prototypes for the functions you will be implementing. You should #include this file from your `ChessMoves.c` file, like so:

```
#include "ChessMoves.h"
```

The "quotes" (as opposed to <brackets>) indicate to the compiler that this header file is found in the same directory as your source, not a system directory.

When working on this program, do not modify `ChessMoves.h` in any way. Do not copy its struct definitions or functional prototypes into your `ChessMoves.c` file. Do not send `ChessMoves.h` to us when you submit your assignment. We will use our own unmodified copy of `ChessMoves.h` when compiling your program.

If you write auxiliary functions ("helper functions") in your `ChessMoves.c` file (which is strongly encouraged!), you should ***not*** add those functional prototypes to `ChessMoves.h`. Our test case programs will not call your helper functions directly, so they do not need any awareness of the fact that your helper functions even exist. (We only list functional prototypes in a header file if we want multiple source files to be able to call those functions.) So, just put the functional prototypes for any helper functions you write at the top of your `ChessMoves.c` file.

Think of `ChessMoves.h` as a bridge between source files. It contains struct definitions that might be used by multiple source files, as well as functional prototypes, but only for functions that might be defined in one source file (such as your `ChessMoves.c` file) and called from a different source file (such as the `testcaseXX.c` files we have provided with this assignment).

There are three structs defined in `ChessMoves.h`, which you will use to implement the required functions for this assignment. They are as follows:

The Game Struct

```
typedef struct Game
{
   char **moves;   // array of algebraic chess notation strings
   int numMoves;   // number of strings in the 'moves' array
} Game;
```

This struct is used to hold several lines of algebraic chess notation describing moves made in a game of chess. It holds a 2D `char` array (an array of strings) called `moves`, and an integer, `numMoves`, that indicates how many strings are in the array. Each string in the `moves` array is guaranteed to be a valid algebraic chess notation string and is guaranteed to be properly terminated with a null sentinel (‘\0’). `numMoves` will always be greater than or equal to zero.

Structs of this type will be processed by your `playTheGame()` function, which is described below in Section 5 (“Function Requirements”). Several test cases included with this assignment give examples of how this struct will be populated before being passed to your `playTheGame()` function.

The Move Struct

```
typedef struct Move
{
   Location from_loc;     // location where this piece is moving from
   Location to_loc;       // location where this piece is moving to
   char piece;            // what type of chess piece is being moved
   short int isCapture;   // whether this move captures another piece
   Color color;           // the color of the piece being moved
} Move;
```

This struct will hold information about each chess move. Your program will process a string of algebraic chess notation, such as “1. e4 e5”, and when it does so, certain pieces of that string will be extracted, interpreted, and used to fill out certain fields in this struct.

The `from_loc` field will hold the location of the chess piece whose movement is being described (a column and row on the chess board; see struct definition below). The `to_loc` field will hold the column and row of the location to which the chess piece is moving. The `piece` field is a single uppercase character denoting what kind of piece is being moved (‘P’ for pawns, ‘R’ for rooks, ‘N’ for knights, ‘B’ for bishops, ‘Q’ for queens, and ‘K’ for kings). The `isCapture` field will be set to 1 if this move results in a piece being captured, or 0 if not. The `color` field is an enumerated datatype (also defined in `ChessMoves.h`; for a description, see below), and will be set to `WHITE` if the chess piece being moved is white, and `BLACK` if the chess piece being moved is black.

Some of these fields will be set by your `parseNotationString()` function. Others cannot be set until you actually go to apply the move to the chess board in your `movePiece()` function, which will call a `findFromLoc()` function to help determine which piece on the board is above to move. These functions are all described in Section 5 of this document (“Function Requirements”).

The Location Struct

```
typedef struct Location
{
   char col;   // the square's column ('a' through 'h')
   int row;    // the square's row (1 through 8)
} Location;
```

This struct holds locations on the chess board, and is used by the `Move` struct defined above. For information on how rows and columns are labeled on a chess board, see “Appendix A: Chess Pieces” and “Appendix B: Algebraic Chess Notation.”

The Color Enum

```
typedef enum { BLACK, WHITE } Color;
```

This enumerated type is used by the `Move` struct to denote the color of the chess piece being moved. The `color` field of that struct can be set to either `BLACK` or `WHITE`. There are examples of how to set enumerated data types in the test cases included with the assignment.

5. Function Requirements

In the source file you submit, `ChessMoves.c`, you must implement the following functions. You may implement auxiliary functions (helper functions) to make these work, as well. Please be sure the spelling, capitalization, and return types of your functions match these prototypes exactly.

In this section, I sometimes refer to `malloc()` when talking about dynamic memory allocation, but you’re welcome to use `calloc()` or `realloc()` instead, if you’re familiar with those functions.

```
char **createChessBoard(void);
```

> **Description:** Dynamically allocate space for a 2D char array with dimensions 8x8, and populate the array with the initial configuration of a chess board as described in “Appendix A: Chess Pieces” (and as shown in the test case output files included with this assignment).
>
> **Output:** This function should not print anything to the screen.
>
> **Return Value:** A pointer to the dynamically allocated 2D array (i.e., the base address of the 2D array), or `NULL` if any calls to `malloc()` fail.

```
char **destroyChessBoard(char **board);
```

> **Description:** Free all dynamically allocated memory associated with `board`. You may assume this function receives a dynamically allocated 2D `char` array with dimensions 8x8.

Output: This function should not print anything to the screen.

Return Value: This function must return `NULL`.

```
void printChessBoard(char **board);
```

Description (and Output): This function takes a pointer to an 8x8 char array and prints the chess board represented by that array using the format described below in the `playTheGame()` function description. This format is also shown explicitly in several of the test case output files included with this assignment.

The printout of the chess board should be preceded by a line of eight equal symbols ('='), and it should also be followed by a line of eight '=' symbols, followed by a blank line. (Again, see the example below in the description of the `playTheGame()` function.)

Return Value: There is no return value. This is a `void` function.

```
char **playTheGame(Game *g);
```

Description: This function should start by printing a chess board with all pieces in their starting positions, using the arrangement described in "Appendix A: Chess Pieces" and the specific format shown in the test case output files included with this assignment.

Specifically, the chess board should include spaces where there are no pieces, and it should have a line of eight equal signs ('=') immediately above and below the board. This should be followed by a blank line. For example:

```
========
RNBQKBNR
PPPPPPPP
           ←
           ←
           ←
           ← Note: There are eight spaces on each of these lines.
pppppppp
rnbqkbnr
========
           ← Note: This is a completely blank line. (No spaces.)
```

(You will want to call `createChessBoard()` from within this function to create a 2D `char` array to represent the chess board, and then call the `printChessBoard()` function to print it to the screen. See above for the descriptions of both of those functions.)

After printing the initial chess board, this function should parse all of the algebraic chess notation strings passed in through the `Game` struct by making calls to the `parseNotationString()` function (described below). Each move described in the notation strings should be applied, sequentially, to the chess board (that is, to the 2D `char` array) by repeatedly calling the `movePiece()` function (described below). Every time a move is applied,

this function should print the resulting board (again, with a line of eight equal signs above and below the board each time, always followed by a blank line).

Output: This should print out the chess board in the manner described above, with precisely the same format shown in the test case output files included with this assignment.

Return Value: Return a pointer to the 2D `char` array created in this function, which should contain the board's final configuration after all the chess moves have been processed.

```
void parseNotationString(char *str, Move *whiteMove, Move *blackMove);
```

Description: This function receives an algebraic chess notation string, `str`, which follows the specification for such strings given in "Appendix B: Algebraic Chess Notation," and two `Move` struct pointers, `whiteMove` and `blackMove`. The function must parse str and extract information about the white and black moves encoded there, and populate all corresponding fields in the structs pointed to by `whiteMove` and `blackMove`.

At the very least, it will always be possible to set the `to_loc`, `piece`, and `isCapture` fields in both `Move` structs. Some strings will contain information about the `from_loc`, as well (specifically if there are multiple chess pieces that could move to the position described, and it becomes necessary to denote which column and/or row the piece making that move is coming from), but sometimes it will be impossible to initialize the column and/or row in a move's `from_loc` field. In the case where a move's `from_loc.col` coordinate cannot be determined from the algebraic chess notation string, you must initialize that move's `from_loc.col` field to 'x' to indicate that the column is currently unknown. Similarly, if a move's `from_loc.row` coordinate cannot be determined, you must initialize that move's `from_loc.row` field to -1.

Special Consideration: If the game ends after White's move, `str` will not contain a string denoting Black's move. In that case, there is no need to initialize any fields in the `blackMove` struct. This situation is described in more detail in "Appendix B: Algebraic Chess Notation" and will only happen in a bonus test case (if at all), so you can ignore this situation if you'd like.

Output: This function should not print anything to the screen.

Return Value: There is no return value. This is a `void` function.

```
void movePiece(char **board, Move *move);
```

Description: This function takes a 2D `char` array representing the current state of the chess board and a `Move` struct indicating a move to be made, and directly modifies the `board` array to represent what the chess board will look like after the move has been made.

You may assume that `move` contains information about a legal move, but it is not guaranteed to have the `from_loc` field initialized. Before applying this move to the board, you will have to search for the location from which the chess piece is moving (if that information is not given in the `move` struct), based on your knowledge of what piece is moving (`move->piece`, which will always be an uppercase letter, whether the piece being moved is black or white), where the

piece is heading (`move->to_loc`), what color the piece is (`move->color`), and whether or not the move is capturing another piece (`move->isCapture`) (which, in the case of a pawn, tells you whether the movement is straight forward or diagonally forward). If there are multiple pieces that could move to the location in question, then the move will necessarily contain `from_loc` information (populated by the `parseNotationString()` function, although it might be just the column or row that is given – not necessarily both) to help you find that piece.

Output: This function should not print anything to the screen.

Return Value: There is no return value. This is a `void` function.

```
void findFromLoc(char **board, Move *move);
```

Description: This function is called from within your `movePiece()` function. It takes a chess board (represented as a 2D `char` array) and a `Move` struct, and finds the piece on the board that is able to make the move described by the struct. It must then populate the move's `from_loc` member with the column and row of the square from which the chess piece is departing.

Please be aware that `move` might contain partial or complete `from_loc` information already, depending on whether that information was included in the algebraic notation string processed by `parseNotationString()` before `move` reached this function.

It is guaranteed that there will be exactly one piece on the board that is able to make the move described by this `move` struct, although you might have to rely on the struct's partially initialized `from_loc` information (if available) in order to determine exactly which piece that is.

Output: This function should not print anything to the screen.

Return Value: There is no return value. This is a `void` function.

```
double difficultyRating(void);
```

Output: This function should not print anything to the screen.

Return Value: A double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
double hoursSpent(void);
```

Output: This function should not print anything to the screen.

Return Value: An estimate (greater than zero) of the number of hours you spent on this assignment.

6. Compilation and Testing (CodeBlocks)

The key to getting multiple files to compile into a single program in CodeBlocks (or any IDE) is to create a project. Here are the step-by-step instructions for creating a project in CodeBlocks, which involves importing `ChessMoves.h`, `testcase01.c`, and the `ChessMoves.c` file you've created (even if it's just an empty file so far).

1. Start CodeBlocks.
2. Create a New Project (*File -> New -> Project*).
3. Choose "Empty Project" and click "Go."
4. In the Project Wizard that opens, click "Next."
5. Input a title for your project (e.g., "ChessMoves").
6. Choose a folder (e.g., Desktop) where CodeBlocks can create a subdirectory for the project.
7. Click "Finish."

Now you need to import your files. You have two options:

1. Drag your source and header files into CodeBlocks. Then right click the tab for **<u>each</u>** file and choose "Add file to active project."

   – or –

2. Go to *Project -> Add Files...*. Browse to the directory with the source and header files you want to import. Select the files from the list (using CTRL-click to select multiple files). Click "Open." In the dialog box that pops up, click "OK."

You should now be good to go. Try to build and run the project (F9).

Note that if you import both `testcase01.c` *and* `testcase02.c`, the compiler will complain that you have multiple definitions for `main()`. You can only have one of those in there at a time. You'll have to swap them out as you test your code.

Yes, constantly swapping out the test cases in your project will be a bit annoying. You can avoid this if you're willing to migrate away from an IDE and start compiling at the command line instead. If you're interested in doing that in Windows, please look around online for instructions on how to make that happen, and see a TA in office hours if you get stuck. Alternatively, you might consider installing Linux on a separate partition of your hard drive. If you take that approach, just be sure to back up your hard drive first.

Note! Even if you develop your code with CodeBlocks on Windows, you ultimately have to transfer it to the Eustis server to compile and test it there. See the following page (Section 7, "Compilation and Testing (Linux/Mac Command Line)") for instructions on command line compilation in Linux.

7. Compilation and Testing (Linux/Mac Command Line)

To compile multiple source files (`.c` files) at the command line:

```
gcc ChessMoves.c testcase01.c
```

By default, this will produce an executable file called `a.out`, which you can run by typing:

```
./a.out
```

If you want to name the executable file something else, use:

```
gcc ChessMoves.c testcase01.c -o ChessMoves.exe
```

...and then run the program using:

```
./ChessMoves.exe
```

Running the program could potentially dump a lot of output to the screen. If you want to redirect your output to a text file in Linux, it's easy. Just run the program using the following command, which will create a file called `whatever.txt` that contains the output from your program:

```
./ChessMoves.exe > whatever.txt
```

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided several sample output files. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt output01.txt
```

If the contents of `whatever.txt` and `output01.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
seansz@eustis:~$ diff whatever.txt output01.txt
seansz@eustis:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same. For example:

```
seansz@eustis:~$ diff whatever.txt output01.txt
1c1
< fail whale :(
---
> Hooray!
seansz@eustis:~$ _
```

8. Getting Started: A Guide for the Overwhelmed

Okay, so, this might all be overwhelming, and you might be thinking, “Where do I even start with this assignment?! I’m in way over my head!”

Don’t panic! There are plenty of TA office hours where you can get help, and here’s my general advice on starting the assignment:

1. First and foremost, start working on this assignment early. Nothing will be more frustrating than running into unexpected errors or not being able to figure out what the assignment is asking you to do on the day that it is due.

2. Secondly, glance through this entire PDF to get a general overview of what the assignment is asking you to do, even if you don’t fully understand each section right away.

3. Thirdly, before you even start programming, read through Appendices A and B to familiarize yourself with the game of chess and the algebraic chess notation you’ll be parsing in this assignment.

4. Once you have an idea of how algebraic chess notation works, open up the test cases and sample output files included with this assignment and trace through a few of them to be sure you have an accurate understanding of how chess notation works.

5. Once you’re ready to begin coding, start by creating a skeleton `ChessMoves.c` file. Add a header comment, add some standard `#include` directives, and be sure to `#include "ChessMoves.h"` from your source file. Then copy and paste each functional prototype from `ChessMoves.h` into `ChessMoves.c`, and set up all those functions return dummy values (either `NULL` or nothing at all, as appropriate).

6. Test that your `ChessMoves.c` source file compiles. If you’re at the command line on a Mac or in Linux, your source file will need to be in the same directory as `ChessMoves.h`, and you can test compilation like so:

   ```
   gcc -c ChessMoves.c
   ```

   Alternatively, you can try compiling it with one of the test case source files, like so:

   ```
   gcc ChessMoves.c testcase01.c
   ```

   For more details, see Section 7, “Compilation and Testing (Linux/Mac Command Line).”

   If you’re using an IDE (i.e., you’re coding with something other than a plain text editor and the command line), open up your IDE and start a project using the instructions above in Section 6, “Compilation and Testing (CodeBlocks)”. Import `ChessMoves.h`, `testcase01.c`, and your new `ChessMoves.c` source file, and get the program compiling and running before you move forward. (Note that CodeBlocks is the only IDE we officially support in this class.)

7. Once you have your project compiling, go back to the list of required functions (Section 5, “Function Requirements”), and try to implement one function at a time. Always stop to compile and test your code before moving on to another function!

8. You’ll *probably* want to start with the `createChessBoard()` function.

   As you work on `createChessBoard()`, write your own `main()` function that calls `createChessBoard()` and then checks the results. For example, you’ll want to ensure that `createChessBoard()` is returning a non-NULL pointer to begin with, and that each index in the array it returns is non-NULL as well. Then try printing out the board. Finally, look through the text cases provided with this assignment to find one that calls `createChessBoard()` explicitly. Run it and check that your output is correct. If not, go through your code (as well as the test case code) line-by-line, and see if you can find out why your output isn’t matching.

9. After writing `createChessBoard()`, I would probably work on the `destroyChessBoard()` and `printChessBoard()` functions, because they’re so closely related. This might require that you spend some time reviewing the course notes on 2D array allocation.

10. Next, I would work on the `parseStringNotation()` function. This one will be quite lengthy. Start by writing your own `main()` that passes simple strings to `parseStringNotation()`. Check that the results it produces are correct. Then start passing more complex strings. If you need guidance on how to call this function and determine whether it’s producing the correct results, look for a test case that calls `parseStringNotation()`, and check whether your program produces the correct output with that test case. If not, trace through it by hand.

11. If you get stuck while working on this assignment, draw diagrams on paper or a whiteboard. Make boxes for all the variables in your program. If you’re dynamically allocating memory, diagram them out and make up addresses for all your variables. Trace through your code carefully using these diagrams.

12. With so many pointers, you’re bound to encounter errors in your code at some point. Use `printf()` statements liberally to verify that your code is producing the results you think it should be producing (rather than making assumptions that certain components are working as intended). You should get in the habit of being immensely skeptical of your own code and using `printf()` to provide yourself with evidence that your code does what you think it does.

13. When looking for a segmentation fault, you should always be able to use `printf()` and `fflush()` to track down the *exact* line you’re crashing on.

14. You’ll need to examine a lot of debugging output. You might want to set up a function that prints debugging strings only when you #define a `DEBUG` value to be something other than zero, so you can easily flip debugging output on and off. (Just be sure to remove your debugging statements before you submit your assignment, so your code is nice and clean and easy for us to read.)

15. When you find a bug, or if your program is crashing on a huge test case, don’t trace through hundreds of lines of code to track down the error. Instead, try to cook up a new `main()` function with a very small test case (as few lines as possible) that directly calls the function that’s crashing. The less code you have to trace through, the easier your debugging tasks will be.

9. Deliverables

Submit a single source file, named `ChessMoves.c`, via Webcourses. The source file should contain definitions for all the required functions (listed above), as well as any auxiliary functions you need to make them work.

Your source file **must not** contain a `main()` function. Do not submit additional source files, and do not submit a modified `ChessMoves.h` header file. Your source file (without a `main()` function) should compile at the command line using the following command:

```
gcc -c ChessMoves.c
```

It must also compile at the command line if you place it in a directory with `ChessMoves.h` and a test case file (for example, `testcase01.c`) and compile like so:

```
gcc ChessMoves.c testcase01.c
```

Be sure to include your name and NID as a comment at the top of your source file.

10. Grading

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

40%	correct output for test cases
35%	implementation details (manual inspection of your code)
5%	`difficultyRating()` is implemented correctly
5%	`hoursSpent()` is implemented correctly
5%	source file is named correctly (`ChessMoves.c`); spelling and capitalization count
10%	adequate comments and whitespace; source includes student name and NID

Note! Your program must be submitted via Webcourses, and it must compile and run on Eustis to receive credit. Programs that do not compile will receive an automatic zero.

Your grade will be based largely on your program's ability to compile and produce the *exact* output expected. Even minor deviations (such as capitalization or punctuation errors) in your output will cause your program's output to be marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program throughly.

Additional points will be awarded for style (proper commenting and whitespace) and adherence to implementation requirements. For example, the graders might inspect your `destroyChessBoard()` function to see whether it is properly freeing up memory.

Start early. Work hard. Good luck!

Appendix A: Chess Pieces

This appendix describes how each piece moves in the game of chess. First, however, we give a very brief overview of the game and describe the initial configuration of the board.

References: Moves That Chess Pieces Can Make (Chess for Dummies Cheat Sheet); Chess (Wikipedia)

Basic Game Overview

Chess is a game for two players: one who manipulates white chess pieces on the chess board, and one who manipulates black chess pieces. Throughout this document, the players are referred to as White and Black, respectively.

Starting with White, the two players take turns moving one piece at a time on the chess board. The board itself is an 8x8 grid of black and white squares. Different pieces are able to make different types of moves (both in terms of the number of squares they can move and the directions in which they can move). A square on the board can only have one piece at any given time. If White moves a piece to a square occupied by a black piece, the black piece is captured and removed from the board (and vice versa if Black moves a piece to a square occupied by a white piece).

The game ends when one of the player's kings is trapped in a situation where it cannot escape being captured by the other player.

Board Configuration

A chess board is an 8x8 grid of alternating black and white squares. When two players sit down across from one another at a chess board, they sit in such a way that the row closest to each player has a white square on the far right.

Pieces are then placed on the board as follows: In the row closest to each player, his or her rooks (also called castles) are placed on the outermost squares. In that same row, a knight is placed next to each rook. Then come the bishops. The king and queen are placed on the innermost squares, with each queen being placed on a square matching its color. (That is, the white queen is placed on a white square, and the black queen is placed on a black square.) The next row is then filled with each player's pawns.

This gives rise to the chess board configuration shown on the following page, in figure A1. In that figure, the letter 'P' is used to represent pawns, 'R' represents rooks, 'N' represents knights, 'B' represents bishops, 'Q' represents queens, and 'K' represents kings. By convention, the board is drawn with the starting rows for White's pieces at the bottom and starting rows for Black's pieces at the top. Rows are then labeled 1 through 8 (from bottom to top), and columns are labeled 'a' through 'h' (from left to right).

Note! To distinguish between white pieces and black pieces any time we print a board (both in this document and in the output produced by your program), we will use uppercase letters to denote the black pieces on a board and lowercase letters to denote the white chess pieces on a board.

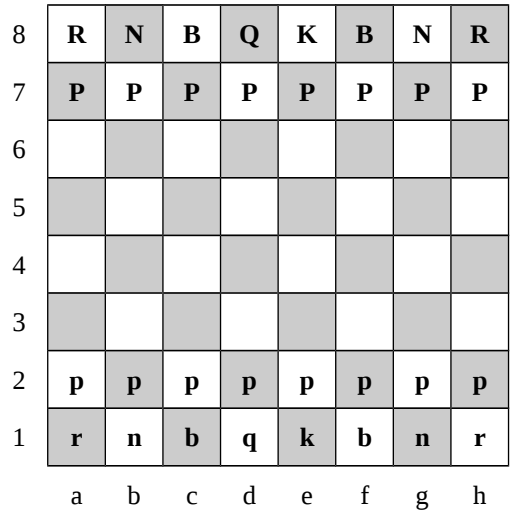


Figure A1. *The initial configuration of an 8x8 chess board. Boards are drawn with the starting rows for White's pieces at the bottom and the starting rows for Black's pieces at the top. For this assignment, we will use uppercase letters to denote black pieces and lowercase letters to denote the white chess pieces any time we print a board.*

Pawns

Basic Pawn Movement:

Pawns always move forward (never backwards) on the chess board. On its first move, each pawn can move one or two squares, as long as it does not jump over any other pieces while doing so. After its first move, a pawn can only move one square forward at a time. (See Figure A2.)

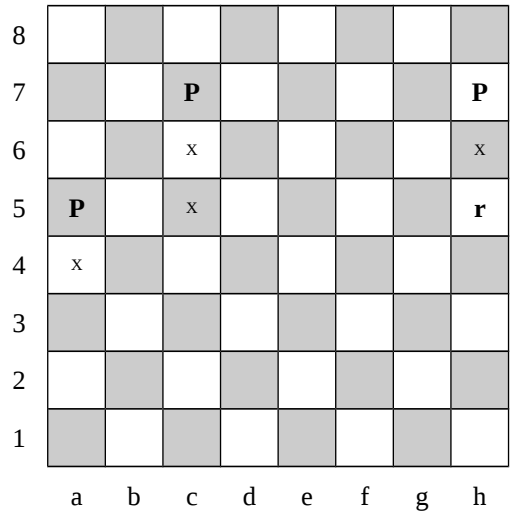


Figure A2. *The spaces where each pawn can move are marked with the letter 'x'. The black pawn on c7 can move to square c6 or c5, since it is making its first move from its starting position. The black pawn on h7 can only move one space forward, since it is blocked by a white rook. The black pawn on a5 may only move forward one space, to a4.*

Pawn Captures:

Pawns are unusual in that their basic movement cannot be used to capture another piece. In order for a pawn to capture another piece, it must move forward diagonally by one square. (See Figure A3.)

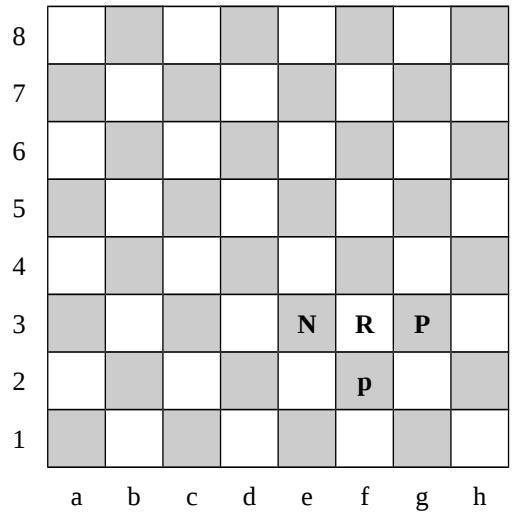


Figure A3. *The white pawn on f2 is blocked by a black rook and therefore cannot move straight forward to position f3. However, it can move diagonally to e3 and capture the black knight there, or it can move diagonally to g3 and capture the black pawn on that square.*

En Passant Captures (Bonus Material):

If a pawn moves forward two spaces on its first move, an opposing pawn is allowed to diagonally attack the space where the pawn would have moved if it had only moved forward one space, thereby capturing the pawn. This is called an "en passant" capture. This can only be done in the move *immediately* after the pawn being captured moved two spaces forward.

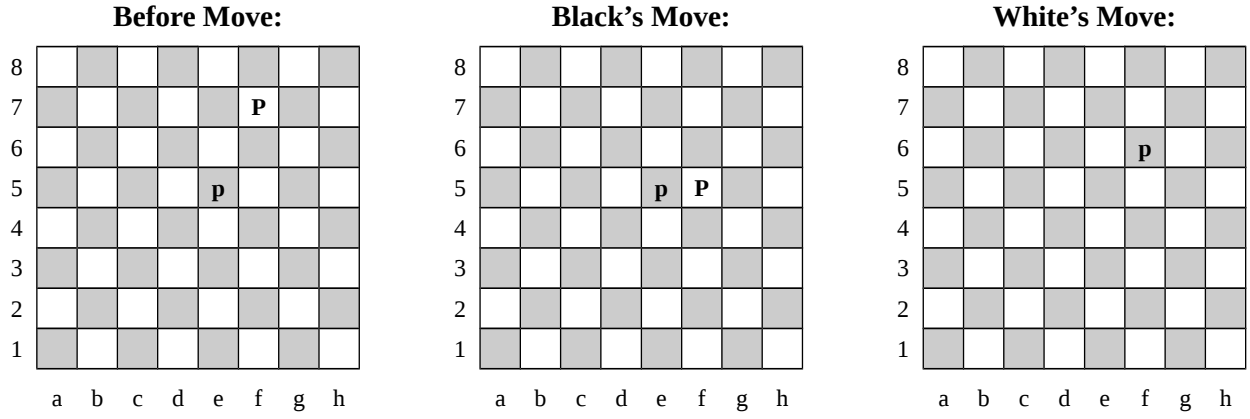


Figure A4. *The black pawn at f7 moves forward two spaces to f5 on its first move. The white pawn at e5 attacks f6 diagonally, which is where the black pawn would have ended up if it had only moved forward one space. The white pawn is attacking the black pawn* en passant *("in passing").*

Pawn Promotion (Bonus Material):

If a pawn reaches the far side of the board (the opposing player's first row), it can be promoted to any piece of the player's choosing, other than a king or a pawn. There is a special notation for promotion (described in Appendix B), and promotions will only happen in bonus test cases for this program.

Note! There is special notation for both *en passant* captures and pawn promotion (described in "Appendix B: Algebraic Chess Notation"). These situations will only happen in bonus test cases for this program, so you actually don't need to worry about them if you don't want to.

Rooks

Basic Rook Movement:

A rook (also called a castle) can move any number of spaces up, down, left, or right on the board, but it cannot jump over pieces. (See Figure A5.)

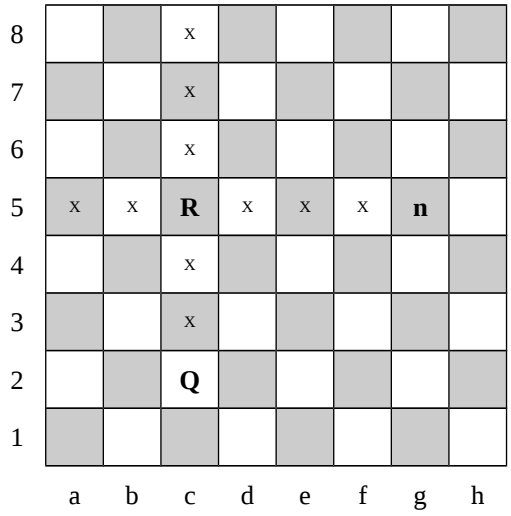


Figure A5. *All the open squares where the black rook on square c5 can move are marked with the letter 'x'. The rook can also move to square g5 and capture the white knight there. Note that the black rook cannot move to c2 and capture the queen, because the queen is also black, and a player cannot capture one of his or her own pieces.*

Castling (Bonus Material):

If one of a player's rooks has never moved, and neither has that player's king, and there are no pieces between said rook and king, then both pieces can move simultaneously in a single move called "castling." In this move, the rook scooches all the way over next to the king, and the king then hops to the other side of the rook. (See Figures A6 and A7 on the following page.)

If the rook being moved was originally closer to the player's king when all the pieces were set up on the board, this is called "kingside castling." If the rook being moved was originally closer to the player's queen, this is called "queenside castling."

Note! There is a special notation for these castling moves (described in "Appendix B: Algebraic Chess Notation"), and castling is guaranteed only to happen in bonus test cases for this assignment (if it happens at all).

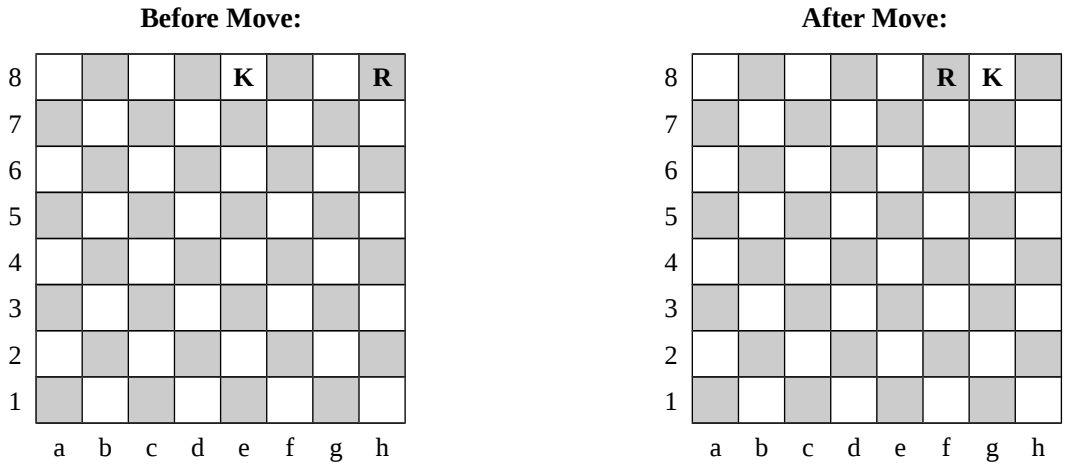


Figure A6. *Kingside castling by Black. The rook moves all the way over next to where the king was positioned, and the king hops to the other side of the rook.*

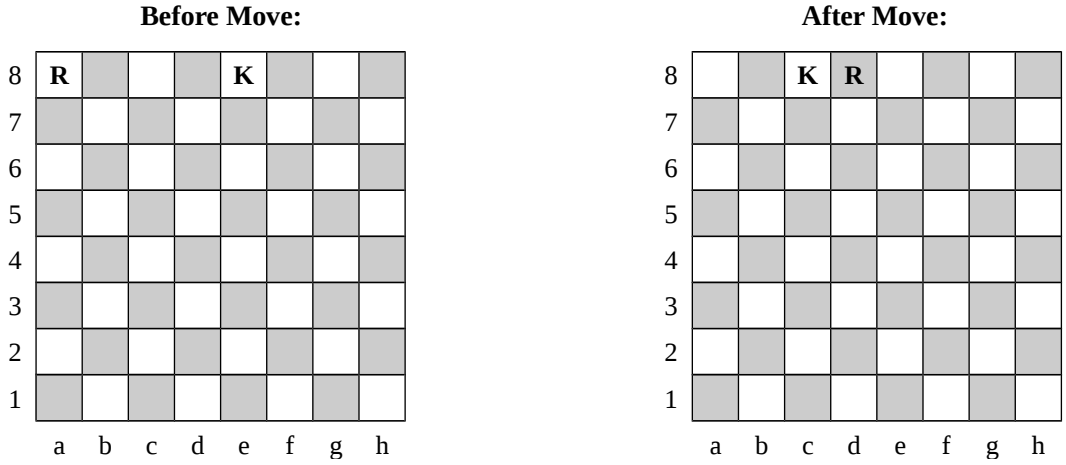


Figure A7. *Queenside castling by Black. The rook moves all the way over next to where the king was positioned, and the king hops to the other side of the rook. This is fancy.*

Knights

Basic Knight Movement:

Knights make L-shaped moves. They can move two spaces up or down, followed by one space to the side (either left or right), or they can move two spaces left or right, followed by one space up or down. Knights are unique in that they can jump over other pieces on the board. (See figure A8 on the following page.)

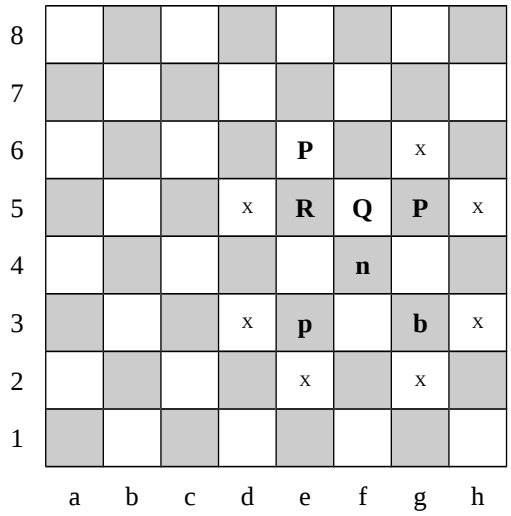


Figure A8. *All the open squares where the white knight on square f4 can move are marked with the letter 'x'. The knight can also capture the black pawn on square e6. Notice that the knight can jump over other pieces on the board.*

Bishops

Basic Bishop Movement:

Bishops can move diagonally any number of spaces. (See Figure A9 below.)

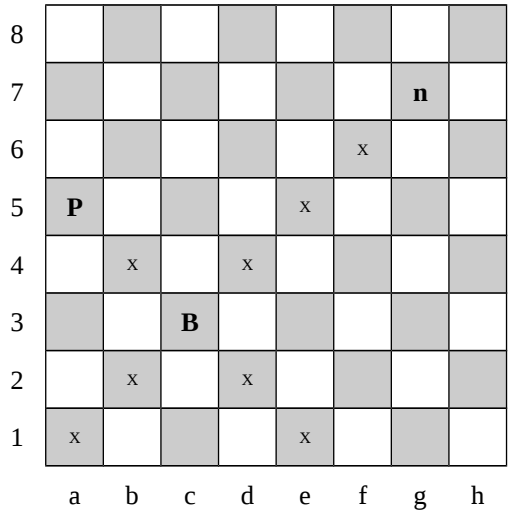


Figure A9. *All the open squares where the black bishop on square c3 can move are marked with the letter 'x'. The bishop can also capture the white knight on square g7. It cannot, however, jump over that knight and move to square h8.*

Queens

Basic Queen Movement:

Queens can move any number of squares in any one of eight directions: up, down, left, right, or any of the four diagonal directions in which a bishop can move. (See Figure A10 below.)

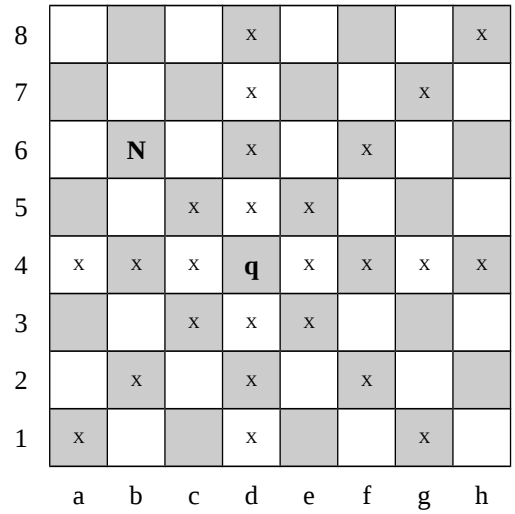


Figure A10. *All the open squares where the white queen on square d4 can move are marked with the letter 'x'. The queen can also capture the black knight on square b6. It cannot, however, jump over that knight and move to position a7.*

Kings

Basic King Movement:

Kings can move one square at a time in any direction. (See Figure A11 below.) They can also engage in castling. (For details on castling, see the section on rooks, above.)

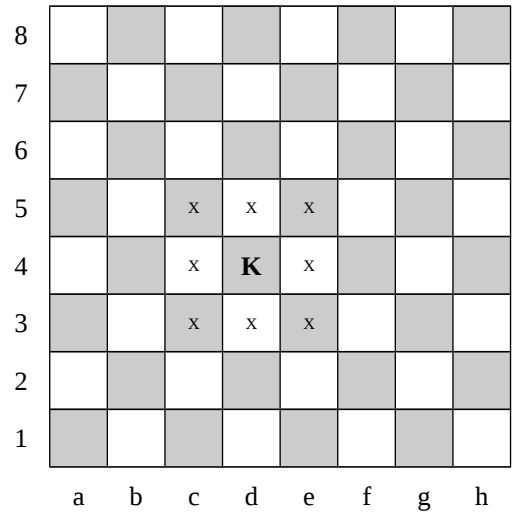


Figure A11. *All the open squares where the black king on square d4 can move are marked with the letter 'x'.*

Appendix B: Algebraic Chess Notation

References: How to Read Algebraic Chess Notation (wikiHow); Algebraic Chess Notation (Wikipedia)

1. Denoting Positions on the Board

To understand algebraic chess notation, one must first understand how to refer to a specific square on an 8x8 chess board. The columns on a chess board are labeled 'a' through 'h', from left to right. The rows are labeled 1 through 8 from bottom to top, where the bottom rows (rows 1 and 2) are where the white pieces are initially placed, and the top rows (rows 7 and 8) are where the black pieces start off the game.

Coordinates for a particular square in this notation system are given as <*column*><*row*>, with no space in between. <*column*> is a single letter from 'a' through 'h', denoting which column a piece is in, and <*row*> is a single integer on the range 1 through 8, denoting which row a piece is in.

For example, in Figure B1 (below), there is a queen at position *c3* and a pawn at position *h6*.

2. Translating Board Positions to 2D Array Coordinates

Arrays in C are referred to as "row major," which means (among other things) that the first index given in *array*[*i*][*j*] refers to the row (that is, row *i*). The second coordinate, therefore, refers to the column (that is, column *j*). Thus, pieces at positions *c3* and *h6* on a chess board represented by a 2D array would be at positions *array*[5][2] and array[2][7], respectively. (See Figure B2, below.)

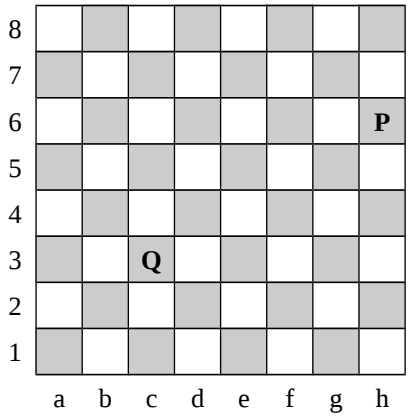


Figure B1: *A chess board with a queen at position c3 and a pawn at position h6.*

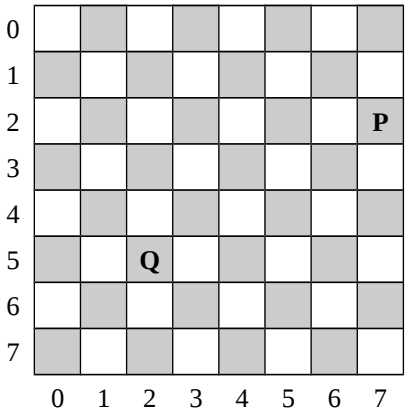


Figure B2: *A 2D array representation of a chess board corresponding to Figure B1. The queen is at array[5][2], and the pawn is at array[2][7].*

In one of the functions you write for this assignment, you will have to take a string containing a coordinate like "c3" and translate it to its corresponding 2D array coordinates (in this case, [5][2]).

3. Denoting Movement

In algebraic chess notation, the moves in a game are denoted by a numbered sequence of strings. Each string begins with an integer, followed by a period and a space, followed by a sequence of characters (without spaces) denoting the movement of a white piece, followed by a single space, followed by a sequence of characters (without spaces) denoting the movement of a black piece.[2] For example:

8. Nd4 Rd3

The number at the beginning of the line is simply the line number. Lines are numbered sequentially, starting from 1. Thus, the "8" in that string indicates that this is the eighth pair of moves made in some game of chess. "Nd4" denotes the movement of a white piece, and "Rd3" denotes the movement of a black piece. (White always moves first.)

The notation for a standard individual move (such as "Nd4" or "Rd3" in the string above) is structured like so:

{*piece*}{*fromCol*}{*fromRow*}{*x*}<*toCol*><*toRow*>

Components in {curly braces} may be left out in some cases, but the components in <angled brackets> will always be present when describing a standard move. There are a few special moves that deviate from this notation, which are described later in this section.

The components that make up these move strings are:

- {*piece*} is a single uppercase character denoting the piece that is moving: 'P' for pawns, 'R' for rooks, 'N' for knights, 'B' for bishops, 'Q' for queens, and 'K' for kings. If the capital letter is omitted, then the piece being moved is a pawn.

  Note: Even though our 2D board arrays contain uppercase letters to represent black pieces and lowercase letters to represent white pieces, the string denoting a move will only use uppercase letters (never lowercase letters) when denoting which piece is being moved.

- <*toCol*> is a single lowercase letter from 'a' through 'h', indicating the column of the square to which this piece is moving.

- <*toRow*> is a single integer from 1 through 8, indicating the row of the square to which this piece is moving.

- If the move results in a piece being captured, a lowercase 'x' will appear between the piece and its destination, in the position labeled {*x*}. (See Example 2 on pg. 25.)

- If the move is ambiguous (e.g., there is more than one piece that could legally make the move being described), the column of the piece being moved will be given in the position labeled {*fromCol*}. This is a single lowercase letter from 'a' through 'h'. (See Example 3 on pg. 26.)

2 Note that there are some exceptions to this, such as when the movement of a white piece causes the end of the game. In that case, there is no need to notate the movement of a black piece. For details on these special situations, see the following page.

- If the move is ambiguous and more than one piece within the same column could legally make the move being described, then the row of the piece being moved will be given instead of the column, in the position labeled {*fromRow*}. This is a single integer from 1 through 8. (See Example 4 on pg. 26.)

- If the move is ambiguous and giving just the column or just the row is not sufficient to indicate which piece is making the move, then both {*fromCol*} and {*fromRow*} will be given, in that order. Incidentally, this is only possible if a pawn has been promoted, and therefore this scenario will only come up in a bonus test case (if at all).

4. Denoting Special Moves

There are a few special moves that warrant their own notation. Strictly speaking, you are not required to handle the notation for any of these moves when processing algebraic notation strings, because these situations will only show up in bonus test cases for this assignment (if at all).

Pawn Promotion:

If a pawn reaches the far side of the board (row 8 for a white pawn, or row 1 for a black pawn), it always gets promoted to another piece of the player's choosing (other than a king or pawn; so, the pawn can be promoted to a queen, bishop, rook, or knight). The piece to which the pawn is promoted is denoted by a single uppercase letter at the end of the move string. For example, a move in which a white pawn reaches position *g8* on the board and is promoted to a knight would be denoted like so: "g8N" (See Example 6 on pg. 27.)

En Passant Captures:

En passant captures are denoted just as normal captures are, except that they end with the string "e.p." For example, the movement of a white pawn to square *f6* via an *en passant* capture would be denoted like so: "xf6e.p." (See Figure A4 on pg. 17.)

Castling:

Kingside castling is denoted "O-O", and queenside castling is denoted "O-O-O". These strings deviate significantly from the notation for a standard move. (Note: Those are capital letter 'O' symbols, not zeros.) For example, the following string has Black castling kingside: "8. c3 O-O" (See Figures A6 and A7 on pg. 19 and Examples 7 and 8 on pg. 27.)

Ending the Game:

If someone wins the game, or if a player resigns, or if there is a draw, the following notation is used: "1-0" appears after the very last move to indicate that White wins, "0-1" is used to indicate that Black wins, and "0.5-0.5" is used to indicate a draw. If White wins or there is a draw, this could lead to a string that does not denote a black move, like so: "44. Qf1 1-0"

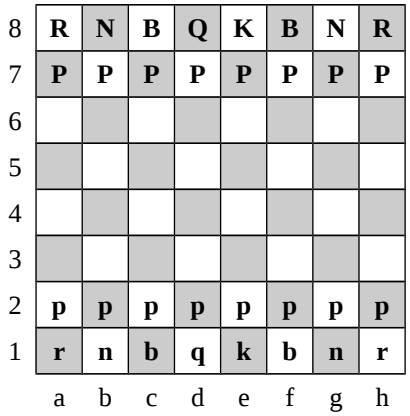
If Black wins, "0-1" is appended to the algebraic notation string like so: "44. Qf1 Qd2 0-1"

In the event of a draw, a string could be formatted as follows: "44. Qf1 Qd2 0.5-0.5"

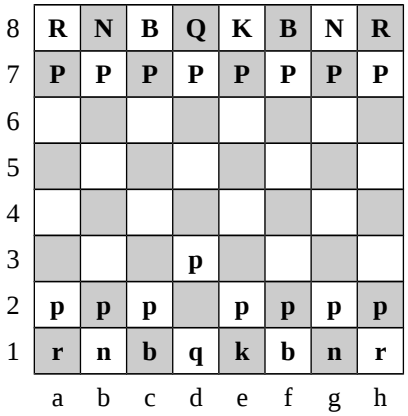
5. Examples

This section provides several examples of how to interpret the notation of a single chess move. In each example, we assume it is a white piece being moved.

Before Move:

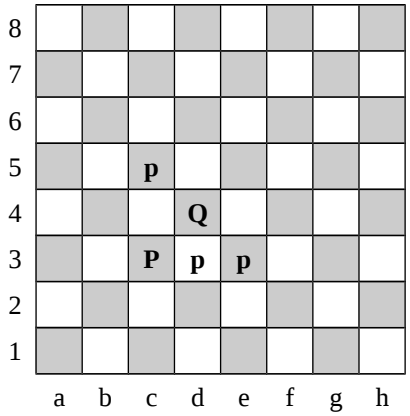


After Move:

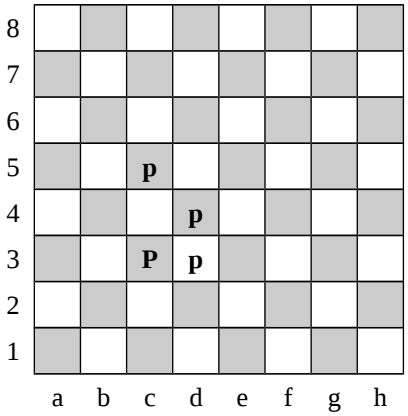


Example 1: *A white pawn moves from d2 to d3. This move can be denoted "Pd3" or simply "d3" (since we can drop the 'P' when a pawn is moving).*

Before Move:



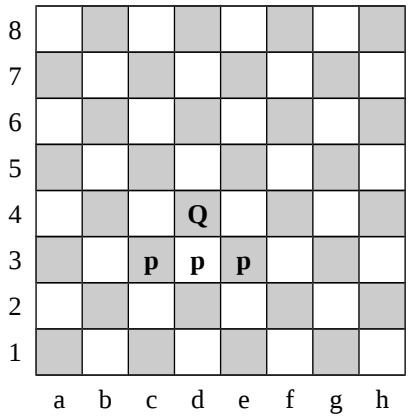
After Move:



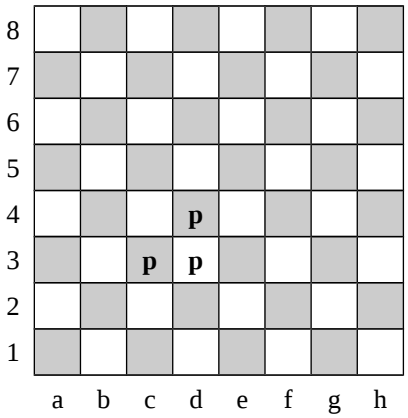
Example 2: *A white pawn moves from e3 to d4, capturing the black queen. Because there is only one pawn on the board that is capable of moving to d4, this move can be denoted "Pxd4" or simply "xd4". Note that the pawn at d3 was incapable of capture the queen, since pawns can only capture pieces by moving diagonally. The pawn at c5 was incapable of capturing the queen because pawns can only move forward, not backwards. The pawn at c3 is a black pawn (which we can tell because it's an uppercase 'P'), and therefore could not attack the black queen. Only one of the white pawns on the board was capable of moving to d4, and so it was not necessary to specify which pawn was making the move.*

Examples continue on the following page…

Before Move:

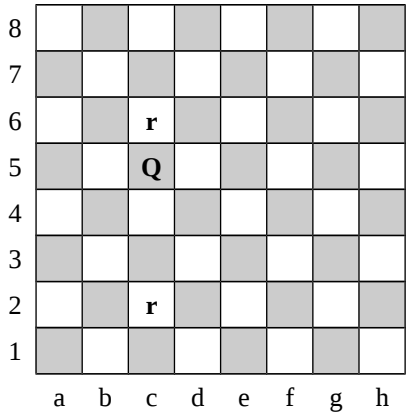


After Move:

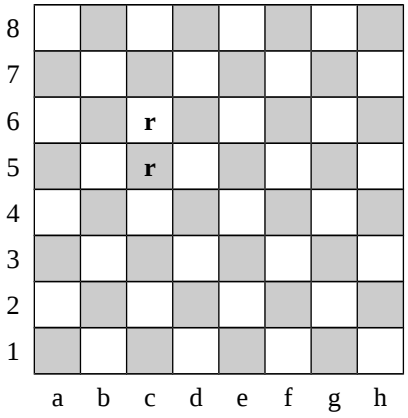


Example 3: *A white pawn moves from e3 to d4, capturing the black queen. Because two white pawns were capable of moving to d4 (the ones on c3 and e3), we cannot simply write "Pxd4." It's essential to give the column of the pawn that made the move. Therefore, this move is denoted either "Pexd4" or simply "exd4".*

Before Move:

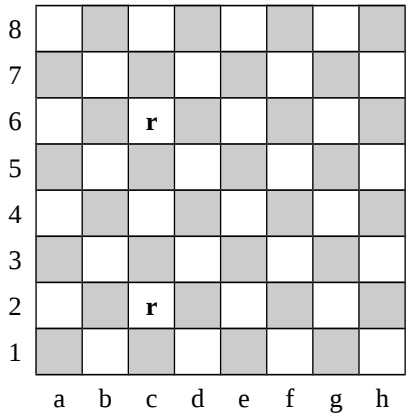


After Move:

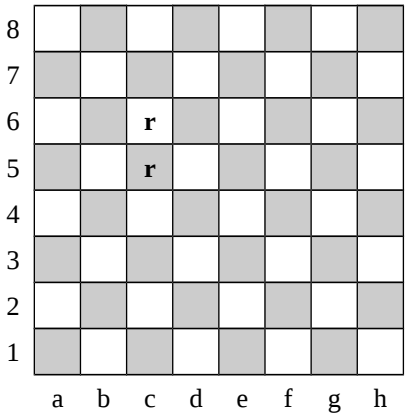


Example 4: *A white rook moves from c2 to c5, capturing the black queen. We cannot simply denote this move as "Rxc5", because both white rooks were able to move to c5. The move must be denoted "R2xc5" to indicate that the rook being moved was the one in row 2.*

Before Move:

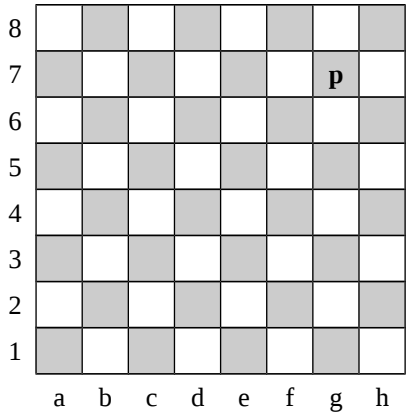


After Move:

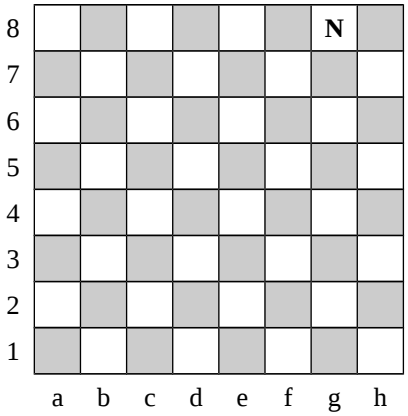


Example 5: *Similar to the previous example, except that no piece is captured. This move is denoted "R2c5".*

Before Move:

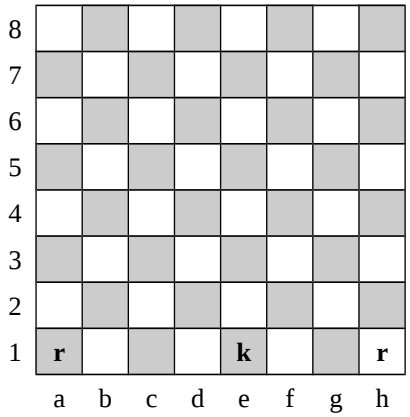


After Move:

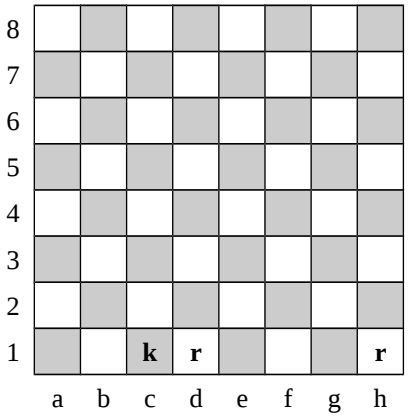


Example 6: *A white pawn moves from g7 to g8 and is promoted to a knight. This move is denoted "g8N". Note that we immediately depict this piece as a knight once it has moved to its new position.*

Before Move:

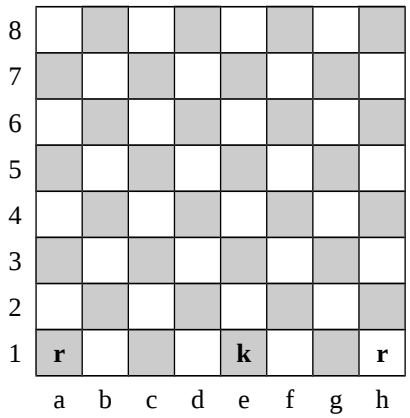


After Move:

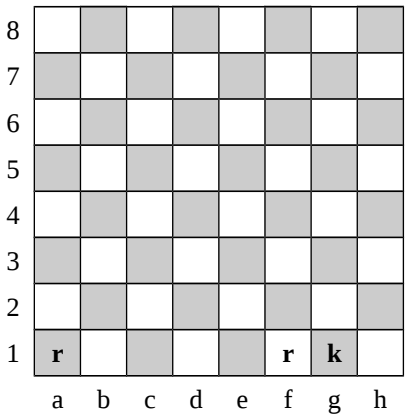


Example 7: *The white king castles queenside. This move is denoted "O-O-O".*

Before Move:



After Move:



Example 8: *The white king castles kingside. This move is denoted "O-O".*