# Project 1:
# Linear Searching For Bobby Fischer

Project due March 8 at 11:59PM via GitHub and the automated grading robot.

You may complete this project as a pair with one other student, but are not required to.

## Overview

In this project, you will develop an implementation of the board game chess, following a general architecture that I have provided for you based on our othello example. Much of the implementation is provided; your major tasks will be to finish `ChessMove` and `ChessBoard`, notably the `GetPossibleMoves`, `ApplyMove`, and `UndoLastMove` methods, and validate that your work passes the tests developed by the class in Homework 2.

## General Architecture

The interfaces `IGameBoard`, `IGameMove`, and `IConsoleView` declare methods that must be implemented by all two-player board games. `ChessBoard`, `ChessMove`, and `ChessView` implement these interfaces to provide a model and view for chess. `Game` implements the controller for the application; you **do not** need to modify anything in the controller.

### ChessBoard:

`ChessBoard` contains member fields to track the state of the game board. You **must** use a one-dimensional array of 32 `byte`s for this purpose, and not any other representation. Each element in the array encodes two adjacent squares, each square given 4 of the 8 bits in the `byte` value. Of those four bits, the first is a player indicator: 0 for player 1 (or empty), and 1 for player 2. The next three bits encode the piece type; in binary:

- 000 for empty
- 001 for pawn
- 010 for rook
- 011 for knight
- 100 for bishop
- 101 for queen
- 110 for king

So the pattern `1011`, equal to 11 (in decimal), represents a black knight, and 0000 represents an empty square. The one-byte value `10101011` represents a black rook next to a black knight, as in rank 8 of the starting board. With an array of 32 of these bytes, we can represent the 64 squares of a chess board.

The methods `GetPieceAtPosition` and `SetPieceAtPosition` should be the **only** methods in your solution that directly read and modify your array. All other operations should utilize those methods to perform their logic. You should start by writing these methods, and using them to set up the initial board state in the ChessBoard constructor.

### ChessMove:

`ChessMove` contains enough information to represent a distinct move on a chess board:

- `StartPosition` and `EndPosition`: the initial position of the piece that is being moved, and the final destination of that piece.
- `MoveType`: an enum of type `ChessMoveType`, flagging moves that are "special", requiring some special logic when applying or undoing:

- `Normal`: most moves. Move the indicated piece to the indicated location, removing any enemy piece that occupies the end position.
- `EnPassant`: perform the en passant, capturing the enemy pawn that is notably *not* occupying the end position.
- `CastleKingSide`: perform a castling move in the direction of the rook that is closer to the king.
- `CastleQueenSide`: perform a castling move in the direction of the rook that is closer to the queen.
- `PawnPromote`: move a pawn using normal movement rules into its final row, then replace the pawn with another piece (rook, bishop, knight, or queen). The move itself must indicate which piece was selected for the promotion; the design of this is up to you.

  The text representation (both for output, and for the user to type) of promotion moves looks like `(start, end, Piece)`, where `Piece` is `Rook`, `Knight`, `Bishop`, or `Queen`.

**ChessConsoleView:**

`ChessConsoleView` handles text-based input and output of chess elements. You need to complete a few missing pieces from this class, and notably need to make sure that the `ParseMove` method constructs pawn promotion moves when needed.

## What You Need To Do

1. **Obtain the starter code for the project**. Follow the link on BeachBoard to the Project 1 Signup page on GitHub. Click the "Accept this assignment" button and follow whatever instructions are required of you to initialize the project. You will end up with a **private** repository named `project1-yourgithubname` in the `csulb-cecs475-2020sp` GitHub organization where the lecture notes are stored. You will then **clone** that repository to your development machine.
2. **Implement the 32-byte array** for the chess board state. Write the `GetPieceAtPosition` method to take a `BoardPosition`, transform that position into an array index and bit mask to strip off the 4 bits associated with that position. Use more bit masks to determine the player and the piece at that position. Then write `SetPieceAtPosition` to do the opposite logic.
3. **Plan an architecture** for getting all attacked positions. A position is *attacked* by a player if the player owns a piece that could take an enemy piece at the position, if there was one there. Each piece type has its own rules for determining which positions it threatens; your design should **at the minimum** use different methods for the different piece types. Implement `GetAttackededPositions` and `PositionIsAttacked` in `ChessBoard`.
4. **Reusing** (which does not mean "copying") the logic for attacked positions, implement `GetPossibleMoves` in `ChessBoard`. Most pieces can move to any position that they attack, except for pawns and kings. Pawns attack diagonally in front, but only move forward. Kings attack one square in all 8 difrections, but cannot move to a position that is attacked by the opponent. **In fact, no move by any piece can leave its own king under attack after the move is applied**. You will need to account for this. This implies that if a king is currently under attack ("in check"), the only possible moves are the ones that result in the king no longer being under attack, whether that involves moving the king, moving another piece in the way of the attacking piece, or capturing the attacking piece. **This is a difficult requirement.**
   (a) Start with simple chess pieces and work your way to harder ones. Note that queens, bishops, and rooks all follow the same rules: attack all squares in a particular set of directions... they differ only in which directions they threaten. A good object-oriented design using inheritance to save code will go a long way here.
   (b) For "simple" moves, the `ChessMove` you create simply has to note the start position and end position of the move.
   (c) Don't worry about castling or en passant at first; implement the basics, then come back.

(d) For “complex” moves, the `ChessMove` created needs to set the `MoveType` of the move to one of the values in `enum ChessMoveType`. This enum will help you undo the move later.

(e) A castling move is indicated by a `ChessMove` from the king’s starting position to **two squares** in the selected castling direction, e.g., to castle king-side (to the right), a move from `e1` to `g1` is created.

(f) An en passant move is indicated by a `ChessMove` from the capturing pawn’s current position to the square that the pawn would have captured, if the opponent’s pawn had not moved two squares forward on their last move.

(g) If a pawn has a move that takes it into the final rank, you will generate moves with a `MoveType` of `PawnPromotion` instead of `Normal`. You must generate **four possible moves** for each of a pawn’s potential moves into the final rank; those four possible moves must represent each of the four valid promotion targets. For example, if a white pawn is at **a7** and square **a8** is unoccupied, then you would generate these moves for that pawn: `(a7, a8, Rook)`, `(a7, a8, Knight)`, `(a7, a8, Bishop)`, `(a7, a8, Queen)`.

i. This implies that `ChessMove` needs to remember more facts about a move than simply “where does it start, where does it end, and what type is it”...

5. Decide how to track auxiliary state for tricky moves like castling and en passant. Castling can only be performed if the king and the rook in question have not moved during the game, and if the king is not in check and does not pass through an attacked position. En passant can only be performed in special circumstances. You may need member variables to keep track of when these moves are valid, and you will need to “save” / remember these variables in your move history in order for undo to work correctly. The design of this is up to you.

6. Write `ApplyMove`. You may assume that any move given to `ApplyMove` is an object that was returned by your own `GetPossibleMoves` method. Using the start and end position of the move, you must update the board state to place the involved pieces in their new positions. You will also need to update any of your miscellaneous board state, like remembering that a king has moved.

7. Implement `UndoLastMove`, which is easier than `GetPossibleMoves`. You will need special logic for some of the tricky moves once again. If you find yourself getting stuck at “how do I know what piece used to be at the end position of this move?”, consider how Othello remembers how many pieces were flipped in each direction by a move.

8. Modify your existing code to keep track of the game’s `CurrentAdvantage`. A chess board’s advantage is given by taking the sum total value of each of white’s pieces and subtracting the sum total of black’s pieces. A pawn is worth 1 value point, knights and bishops worth 3 points, a rook worth 5 points, a queen worth 9 points, and a king worth $\infty$ (but since kings are never captured, this value is ignored). Hint: you want to modify this value each time a move is made or undone, but **without** recounting every square on the board.

9. Write `IsCheck`, `IsCheckmate`, and `IsStalemate`. At most one of these properties can be true at any time.

10. Incorporate the **50-move rule** into your solution. This rule states that a game ends in a stalemate if no piece has been captured and no pawn has moved in the most recent **50 turns by each player** (100 `ApplyMove` calls total). The ChessBoard property `DrawCounter` should return the current progress towards this rule, with each move that is applied contributing to the `DrawCounter` if it does not move a pawn and is not a capture, and resetting it to 0 otherwise.

## Game Rules

See the Wikipedia article titled **Chess** for an overview of how the game is played. We will note the following rules due to their difficulty of implementation:

1. Each move is represented by the **user** by a twist on “algebraic notation”, in which a move indicates its starting position and its ending position. Positions are denoted using letters and numbers; the

lowest row on the board is row `1`, the next up is row `2`, etc.; the leftmost column is column `a`, the next is `b`, etc. Thus, position `a1` is the lower left corner of the board, and `h8` is the upper right corner. **Internally**, positions are represented by `BoardPosition` objects, which view the lower left corner as **row 7, column 0**. Note this difference in the **user's view** vs. the **system's view**.

2. Castling can only be performed if the king and the rook in question have not moved during the game. Additionally, the king cannot be in check, and cannot pass through a position that is under threat.
3. A pawn that is still in its original position can make two different moves, if able: one square forward, or two squares forward, provided there are no enemy pieces directly in the way.
4. En passant is the only move in which a piece captures another piece but does not move onto the square occupied by the captured piece. It is only allowed if the previous move was by a pawn advancing two positions from its starting position, and there is an enemy pawn that could have captured it if the pawn had moved only one position forward.

## Test Cases

Once I have them all organized, I will distribute a set of test cases that your peers turned in for Homework 2. These test cases, plus others, will be run against your code when you turn it in, so you should run them yourself as you make progress with the project. The tests will be loosely organized based on the pieces and features they involve, so you will be able to run only the pawn test cases if you have only finished pawn logic.

## Project Organization

Follow the project layout in the given starter code. You may add files to the `Cecs475.BoardGames.Chess.Model` project, but should not need to elsewhere.

## Submitting For Grading

When you are passing all your test cases locally, and have otherwise tested your application thoroughly by hand, you can then request a grade from the class grading robot. This process is automatically triggered **every time you push your changes to GitHub**. Any time a commit is pushed to GitHub, the grading robot will clone a copy of your project, download the secret test cases, and then run your code against the test cases. **You should receive an email within 10 minutes of pushing your code, and should email me if you do not receive a message within 30 minutes.** The email will detail the results of your submission: a congratulatory message if you passed, or the output of the `dotnet test` command if you failed. If you fail the submission, you will need to fix the errors reported and then push again until you pass. **You can submit as many times as necessary to pass the test cases; there is no penalty for the number of submissions made.**

**Note**: the grading robot will send an email to whatever address is associated with your GitHub account.