40 Monaro Cr

2014

SDD Project Part II

Chess Game

Andrew M Hall

Table of Contents

[Rational for the use of an Event Driven Language 5](#_Toc381873957)

[Structure Chart 6](#_Toc381873958)

[Screen Design 7](#_Toc381873959)

[Global Variables 8](#_Toc381873960)

[Subroutines 9](#_Toc381873961)

[Initialise\_Board 9](#_Toc381873962)

[Pseudocode 9](#_Toc381873963)

[Test Data/ Testing 9](#_Toc381873964)

[Arguments to be passed 9](#_Toc381873965)

[fill\_board 10](#_Toc381873966)

[Data Dictionary 10](#_Toc381873967)

[Pseudocode 10](#_Toc381873968)

[Test Data/ Testing 11](#_Toc381873969)

[Arguments to be Passed 11](#_Toc381873970)

[display\_board 12](#_Toc381873971)

[Data Dictionary 12](#_Toc381873972)

[Pseudocode 12](#_Toc381873973)

[Test Data/ Testing 12](#_Toc381873974)

[Arguments to be Passed 12](#_Toc381873975)

[fill\_picbox\_array 13](#_Toc381873976)

[Pseudocode 13](#_Toc381873977)

[Test Data/Testing 14](#_Toc381873978)

[Arguments to be Passed 14](#_Toc381873979)

[make\_move 15](#_Toc381873980)

[Data Dictionary 15](#_Toc381873981)

[Pseudocode 15](#_Toc381873982)

[Test Data/ Testing 16](#_Toc381873983)

[Arguments to be Passed 16](#_Toc381873984)

[get\_move 17](#_Toc381873985)

[Data Dictionary 17](#_Toc381873986)

[Pseudocode 17](#_Toc381873987)

[Test Data 17](#_Toc381873988)

[Arguments to be Passed 18](#_Toc381873989)

[check\_move 19](#_Toc381873990)

[Data Dictionary 19](#_Toc381873991)

[Pseudocode 19](#_Toc381873992)

[Test Data 19](#_Toc381873993)

[Arguments to be Passed 20](#_Toc381873994)

[execute 21](#_Toc381873995)

[Data Dictionary 21](#_Toc381873996)

[Pseudocode 21](#_Toc381873997)

[Test Data 21](#_Toc381873998)

[Arguments to be Passed 21](#_Toc381873999)

[update\_score 22](#_Toc381874000)

[Data Dictionary 22](#_Toc381874001)

[Pseudocode 22](#_Toc381874002)

[Test Data 23](#_Toc381874003)

[Arguments to be Passed 23](#_Toc381874004)

[update\_board 24](#_Toc381874005)

[Data Dictionary 24](#_Toc381874006)

[Pseudocode 24](#_Toc381874007)

[Test Data 24](#_Toc381874008)

[Arguments to be Passed 24](#_Toc381874009)

[scoring 25](#_Toc381874010)

[Data Dictionary 25](#_Toc381874011)

[Pseudocode 25](#_Toc381874012)

[Test Data 25](#_Toc381874013)

[Arguments to be Passed 25](#_Toc381874014)

[get\_high\_scores 26](#_Toc381874015)

[Data Dictionary 26](#_Toc381874016)

[Pseudocode 26](#_Toc381874017)

[Test Data 26](#_Toc381874018)

[Arguments to be Passed 26](#_Toc381874019)

[write\_file 27](#_Toc381874020)

[Data Dictionary 27](#_Toc381874021)

[Pseudocode 27](#_Toc381874022)

[Test Data 27](#_Toc381874023)

[Arguments to be Passed 27](#_Toc381874024)

[disp\_high\_scores 28](#_Toc381874025)

[Data Dictionary 28](#_Toc381874026)

[Pseudocode 28](#_Toc381874027)

[Test Data 28](#_Toc381874028)

[Arguments to be Passed 28](#_Toc381874029)

[Functions 29](#_Toc381874030)

[check\_pawn 29](#_Toc381874031)

[Data Dictionary 29](#_Toc381874032)

[Pseudocode 29](#_Toc381874033)

[Test Data 30](#_Toc381874034)

[Arguments to be Passed 31](#_Toc381874035)

[check\_rook 32](#_Toc381874036)

[Data Dictionary 32](#_Toc381874037)

[Pseudocode 32](#_Toc381874038)

[Test Data 33](#_Toc381874039)

[Arguments to be Passed 33](#_Toc381874040)

[check\_horse 34](#_Toc381874041)

[Data Dictionary 34](#_Toc381874042)

[Pseudocode 34](#_Toc381874043)

[Test Data 34](#_Toc381874044)

[Arguments to be Passed 35](#_Toc381874045)

[check\_bishop 36](#_Toc381874046)

[Data Dictionary 36](#_Toc381874047)

[Pseudocode 36](#_Toc381874048)

[Test Data 37](#_Toc381874049)

[Arguments to be Passed 37](#_Toc381874050)

[get\_increment 38](#_Toc381874051)

[Data Dictionary 38](#_Toc381874052)

[Pseudocode 38](#_Toc381874053)

[Test Data 38](#_Toc381874054)

[Arguments to be Passed 39](#_Toc381874055)

[check\_queen 40](#_Toc381874056)

[Data Dictionary 40](#_Toc381874057)

[Pseudocode 40](#_Toc381874058)

[Test Data 41](#_Toc381874059)

[Arguments to be Passed 41](#_Toc381874060)

[check\_king 42](#_Toc381874061)

[Data Dictionary 42](#_Toc381874062)

[Pseudocode 42](#_Toc381874063)

[Test Data 43](#_Toc381874064)

[Arguments to be Passed 43](#_Toc381874065)

[sort\_scores 44](#_Toc381874066)

[Data Dictionary 44](#_Toc381874067)

[Pseudocode 44](#_Toc381874068)

[Test Data 45](#_Toc381874069)

[Arguments to be Passed 45](#_Toc381874070)

[Data Structures 46](#_Toc381874071)

[Arrays 46](#_Toc381874072)

[Files 47](#_Toc381874073)

# Rational for the use of an Event Driven Language

The use of an event driven language for this project is appropriate because the game chess, uses user input to effect the order in which the logic flows. These events are such things as clicking on a tile, pressing a button to display high scores and starting the game. Event driven programming languages are the dominant language used with GUI’s and since chess game utilises a GUI, an event driven language is appropriate.

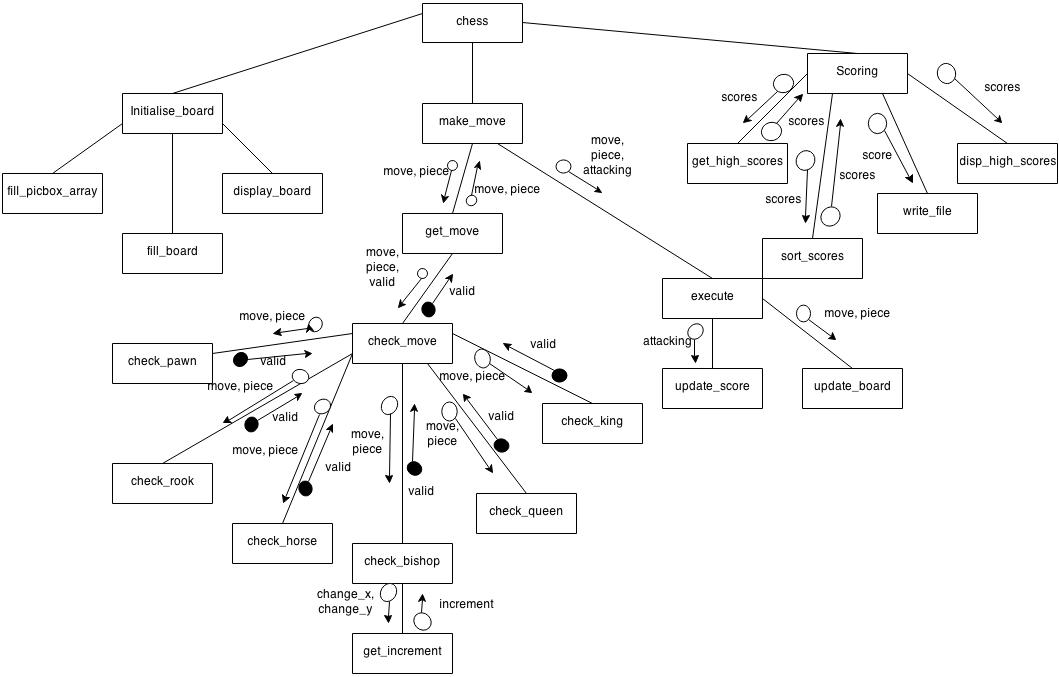
The use of Visual Basic to implement “Chess Game” is good because:

* Visual Basic is an event driven language
* Visual Basic is easy read
* Visual Basic uses a .NET Framework that is developed to make simple programs easy to make and compile
* The .NET Framework allows easy form creation compared to other languages
* Visual Basic has a wide range of inbuilt functions and methods to allow easy programming
* Visual Basic is updated regularly by Microsoft and is thus Up to Date
* Detailed online documentation of Visual Basic is available (MSDN)
* Code syntax is very similar to that of pseudocode

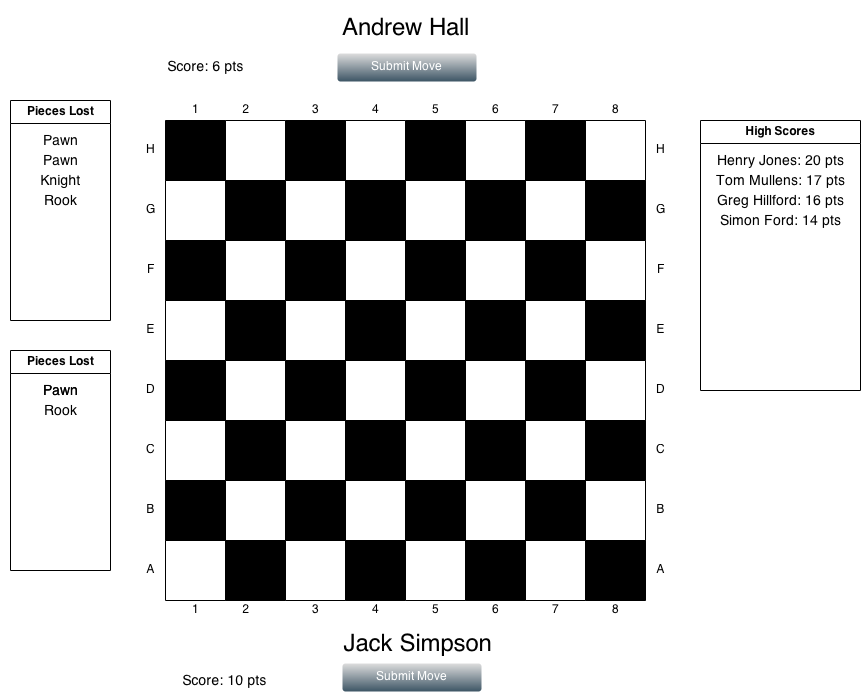
The use of Visual Basic to implement “Chess Game” is bad because:

* Visual Basic code is often far longer than equivalent code in other languages
* Visual Basic has generally poor array manipulation compared to other languages
* Its lengthy code sentence structure often makes otherwise simple tasks feel more complex. This is compared to languages with more truncated syntax structure.

# Structure Chart



# Screen Design



# Global Variables

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| click\_counter | integer | global | X | stores the number of clicks that have been made by the user(s) |
| from | integer | global | XX | Stores the name of the tile that the player is moving his piece from |
| to | integer | global | XX | Stores the name of the tile that the player is moving his piece to |
| p1score | integer | global | XXX | Stores the score of the first player |
| p2score | integer | global | XXX | Stores the score of the second player |
| turn | string | global | SS | Stores a string indicating who's turn it is. e.g. "p1" |

# Subroutines

## Initialise\_Board

### Pseudocode

BEGIN Initialise\_Board()

fill\_picbox\_array()

fill\_board(board)

display\_board()

END Initialise\_Board

### Test Data/ Testing

This sub takes no data and can only be tested once fill\_picbox\_array(), fill\_board() and display\_board() have been developed and thoroughly tested. Once this has been done, the sub can be tested to see if it outputs the filled board in the correct order

### Arguments to be passed

* Initialise\_board accepts no arguments
* It will output a board in the starting setup

## fill\_board

### Data Dictionary

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| Pic | string | local | SS | Stores a 2 letter acronym that defines each piece, e.g. "wr" (white rook) |
| X | integer | local | XX | Stores the x co-ordinate of the table when the for…next loop is filling the table |
| Y | integer | local | XX | Stores the y co-ordinate of the table when the for…next loop is filling the table |

### Pseudocode

BEGIN fill\_board()

pic = ""

FOR x = 1 to 8

FOR y = 1 to 8

Pic = “”

REM this resets pic after each iteration

IF y <= 2 THEN

REM these tiles will be occupied by the white pieces

pic = pic + "w"

END IF

IF y => 7 THEN

REM these tiles will be occupied by the black pieces

pic = pic + "b"

END IF

IF y = 2 or y = 7 THEN

REM these tiles will be occupied by the pawns

pic = pic + "p"

END IF

IF y = 1 or y = 8 THEN

REM these tiles will be occupied by the other pieces

CASEWHERE x

REM this will append the correct prefix too pic variable

Case 1 or 8 : pic = pic + "r"

Case 2 or 7 : pic = pic + "h"

Case 3 or 6 : pic = pic + "b

Case 4 : pic = pic + "q"

Case 5 : pic = pic + "k"

END CASE

IF y = 8 THEN

REM this code swaps the black king and queen

IF x = 4 THEN

pic = "bq"

END IF

IF x = 5 THEN

pic = "bk"

END IF

END IF

END IF

board(x, y) = pic

NEXT y

NEXT x

END fill\_board

### Test Data/ Testing

Fill\_board() takes no data but It can be tested by declaring a global 2D array board(8, 8) THEN calling fill\_board() and outputting the array board into a list box.

Declare global board(8, 8) as integer

BEGIN test on click

fill\_board()

FOR x = 1 to 8

FOR y = 1 to 8

Output board(x, y) into listbox

NEXT y

NEXT x

END test

The expected output is an array filled like this:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Wr | Wh | wb | Wq | wk | wb | Wh | wr |
| Wp | Wp | wp | Wp | wp | wp | Wp | wp |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| Bp | Bp | bp | Bp | bp | bp | Bp | bp |
| Br | Bh | bb | Bk | bq | bb | Bh | br |

### Arguments to be Passed

* fill\_board takes no arguments
* It will output the board array filled with the starting combination of pieces

## display\_board

### Data Dictionary

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| X | integer | Local | XX | Stores the x co-ordinate of the table when the for…next loop is displaying the pictures in the table |
| Y | integer | Local | XX | Stores the y co-ordinate of the table when the for…next loop is displaying the pictures in the table |

### Pseudocode

BEGIN display\_board()

REM will output the pictures into the board based on the acronyms REM contained in board

FOR x = 1 to 8

FOR y = 1 to 8

IF board(x, y) <> "" THEN

REM if the element = “”, tile is empty

picbox(x, y).image="filepath\"+board(x, y)+".jpg"

REM will place the pic onto the screen

END IF

NEXT y

NEXT x

END display\_board

### Test Data/ Testing

This sub can be tested once fill\_board() and fill\_picbox\_array() have been tested. To test this sub will require a driver that is identical to initialise\_board(). So once this sub is done, one can test initialise\_board() by calling the sub and seeing if the output is what is required. That is a chess board output onto picture boxes

### Arguments to be Passed

* display\_board takes no arguments
* will output the board into the picture boxes

## fill\_picbox\_array

### Pseudocode

BEGIN fill\_picbox\_array()

picbox(1, 1) = tile1

picbox(1, 2) = tile2

picbox(1, 3) = tile3

picbox(1, 4) = tile4

picbox(1, 5) = tile5

picbox(1, 6) = tile6

picbox(1, 7) = tile7

picbox(1, 8) = tile8

picbox(2, 1) = tile9

picbox(2, 2) = tile10

picbox(2, 3) = tile11

picbox(2, 4) = tile12

picbox(2, 5) = tile13

picbox(2, 6) = tile14

picbox(2, 7) = tile15

picbox(2, 8) = tile16

picbox(3, 1) = tile17

picbox(3, 2) = tile18

picbox(3, 3) = tile19

picbox(3, 4) = tile20

picbox(3, 5) = tile21

picbox(3, 6) = tile22

picbox(3, 7) = tile23

picbox(3, 8) = tile24

picbox(4, 1) = tile25

picbox(4, 2) = tile26

picbox(4, 3) = tile27

picbox(4, 4) = tile28

picbox(4, 5) = tile29

picbox(4, 6) = tile30

picbox(4, 7) = tile31

picbox(4, 8) = tile32

picbox(5, 1) = tile33

picbox(5, 2) = tile34

picbox(5, 3) = tile35

picbox(5, 4) = tile36

picbox(5, 5) = tile37

picbox(5, 6) = tile38

picbox(5, 7) = tile39

picbox(5, 8) = tile40

picbox(6, 1) = tile41

picbox(6, 2) = tile42

picbox(6, 3) = tile43

picbox(6, 4) = tile44

picbox(6, 5) = tile45

picbox(6, 6) = tile46

picbox(6, 7) = tile47

picbox(6, 8) = tile48

picbox(7, 1) = tile49

picbox(7, 2) = tile50

picbox(7, 3) = tile51

picbox(7, 4) = tile52

picbox(7, 5) = tile53

picbox(7, 6) = tile54

picbox(7, 7) = tile55

picbox(7, 8) = tile56

picbox(8, 1) = tile57

picbox(8, 2) = tile58

picbox(8, 3) = tile59

picbox(8, 4) = tile60

picbox(8, 5) = tile61

picbox(8, 6) = tile62

picbox(8, 7) = tile63

picbox(8, 8) = tile64

END fill\_picbox\_array

### Test Data/Testing

This sub requires 64 pictureboxes placed in a grid with the names “tile” + n (where n Is an incrementing integer between 1 and 64 inclusive). Once the required pictureboxes have been created (either manually or programmatically), one can test this sub by calling it, and THEN outputting a test picture into every picturebox to see if it works

Declare global picbox(8, 8) as picturebox

BEGIN test on click

REM this will test the fill\_picbox\_array sub

fill\_picbox\_array()

FOR x = 1 to 8

FOR y = 1 to 8

Picbox(x, y).image = “filepath\testimage.jpg”

NEXT y

NEXT x

END test

The expected output is a 64 element grid of picture boxes with the same test image in each picture box.

### Arguments to be Passed

* fill\_picbox\_array takes no arguments
* will output the picbox array filled with the 64 picture boxes that make up the board

## make\_move

### Data Dictionary

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| attacking | String | Local | SS | Stores the acronym of the piece that is being attacked, if there is no piece being attacked THEN it is "" |
| move | Array (integer) | Local | (2, 2) | Stores the x, y co-ordinates of the from and to variables for each move |
| piece | String | Local | SS | Stores the acronym of the piece being moved |
| valid | Boolean | Local | True, False | Stores whether a particular move is valid |

### Pseudocode

BEGIN make\_move()

attacking = ""

move = (2, 2)

REM move will contain two sets of (x, y) co-ordinates that define REM where the move is from, and where it is too

piece = ""

REM a string that contains the acronym of the piece moved e.g. wq

valid = False

get\_move(move)

check\_move(piece, move)

attacking = board(move(2,1), move(2,2))

IF Not valid THEN

print messagebox("Invalid Move, please Re-enter")

click\_counter = 0

IF turn = "p1" THEN

REM switching them here means that when they are

REM switched again, it is back to the same player

turn = "p2"

ELSE

turn = "p1"

END IF

ELSE

execute(move, piece, attacking)

END IF

END make\_move

### Test Data/ Testing

This sub can only be tested once get\_move(), make\_move() and execute() have been created and tested or filled with stubs. Once all of these subs have been created, you could test it by declaring to and from as test values and calling the initialise\_board() sub before calling make\_move() and testing the result.

BEGIN test on click

REM is designed to test make\_move

REM assumes that initialise\_board() has been thoroughly tested

to = test value

from = test value

initialise\_board()

make\_move()

END test

### Arguments to be Passed

* make\_move takes no arguments
* make\_move will get and execute a move based on global variables to and from

## get\_move

### Data Dictionary

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| Length | integer | Local | X | Stores the length of the name of the tile which has been clicked |
| Move | Array (integer) | Local | (2, 2) | Stores the x, y co-ordinates of the from and to variables for each move |
| Piece | string | Local | SS | Stores the acronym of the piece being moved |
| Temp | integer | Local | X | Stores the number at the end of the name of the tile that has been clicked |

### Pseudocode

BEGIN get\_move(ByRef move, ByRef piece)

length = len(from) REM gets the length of the string,

REM length - 4 will be only the digits of the name

REM will take the last digit(s) from the name "from", uses

REM the fact that the name will

REM always be "tile" + n (where n in an integer)

temp = Int(Right(from, length - 4))

REM e.g. "tile23" becomes 23

REM this will turn numbers into co-ordinates

REM is equivalent to: =CONCATENATE(MOD(temp+1,8)-1,", REM",(INT((temp+1)/8)-1)) in excel

move(1, 1) = Int((temp + 1) / 8) - 1

move(1, 2) = ((temp + 1) mod 8) - 1

length = len(to)

temp = Int(Right(from, length - 4))

move(2, 1) = Int((temp + 1) / 8) -1

move(2, 2) = ((temp + 1) mod 8) - 1

piece = board(move(1,1), move(1,2))

REM move now contains the two co-ordinates of the move

REM e.g. if move contains [(3, 4), (4, 5)] it would

REM represent a move from point (3, 4)

REM to point (4, 5). therefore move is declared as a string

REM piece now contains the acronym of the piece moving

END get\_move

### Test Data

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Value | Expected Output | Reason for inclusion |
| To | “tile9” | Move = [(1,2), (1, 3)  Piece = “wp” | To test a simple move |
| From | “tile17” |
| To | “tile2” | Move = [(2,1), (2,3)]  Piece = “wh” | To test a harder case |
| From | “tile18” |

### Arguments to be Passed

* get\_move accepts two arguments and uses two global variables
  + move: a 2D array or integers [from(x, y), to(x, y)]
  + piece: a string of length 2 first letter = colour, second letter = piece. e.g. “bk”
  + to: global string that is the name of the tile clicked e.g. “tile6”
  + from: global string that is the name of the tile clicked
* get\_move returns a 2D array and a string, being the result of the tiles clicked and the board

## check\_move

### Data Dictionary

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| Move | Array (integer) | Local | (2, 2) | Stores the x, y co-ordinates of the from and to variables for each move |
| Piece | String | Local | SS | Stores the acronym of the piece being moved |
| Valid | Boolean | Local | True, False | Stores whether a particular move is valid |

### Pseudocode

BEGIN check\_move(piece, move, ByRef valid):

IF turn = "p1" and Left(piece) <> "w" or turn = "p2" and Left(piece) <> "b"

REM checks that the right player is moving his/her piece

valid = False

REM check's that the player is moving the correct piece

END IF

IF move(1,1) = move(2,1) and move(1,2) = move(2,2) THEN

REM checks that player is not double clicking and not moving

Valid = False

END IF

CASEWHERE Right(piece)

REM gets only the Righter most letter or the piece string

REM passing the piece variable is necessary because the colour of REM the piece defines where it can move in some cases

case "p" : valid = check\_pawn(piece, move)

case "r" : valid = check\_rook(piece, move)

case "h" : valid = check\_horse(piece, move)

case "b" : valid = check\_bishop(piece, move)

case "q" : valid = check\_queen(piece, move)

case "k" : valid = check\_king(piece, move)

End CASE

END check\_move

### Test Data

Check\_move takes two arguments and returns valid based on the results.

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Value | Expected Output | Reason for Inclusion |
| Move | [(1,2), (1,3)] | Valid = True | To test a simple true case |
| Piece | “wp” |
| Move | [(4,4), (5,6)] | Valid = False | To test when piece is empty |
| Piece | “” |
| Move | [(2, 1), (2, 5)] | Valid = False | To test an invalid move |
| Piece | “wh” |
| Move | [(4,3), (4,3)] | Valid = False | To test that players can’t double click |
| Piece | “bb” |

### Arguments to be Passed

* check\_move takes 2 arguments and returns 1
  + piece: a string of length 2 first letter = colour, second letter = piece
  + move: a 2D array or integers [from(x, y), to(x, y)]
* check\_move returns valid, a Boolean value indicating whether a move is valid

## execute

### Data Dictionary

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| attacking | String | Local | SS | Stores the acronym of the piece that is being attacked, if there is no piece being attacked THEN it is "" |
| Move | Array (integer) | Local | (2, 2) | Stores the x, y co-ordinates of the from and to variables for each move |
| Piece | String | Local | SS | Stores the acronym of the piece being moved |

### Pseudocode

BEGIN execute(move, piece, attacking)

IF attacking <> "" THEN

update\_score(attacking)

END IF

IF Right(board(move(2,1), move(2,2))) = "k" THEN

REM ends the game if a king is taken

game\_ended = True

IF left(piece) = "w" THEN

REM returns the winning player

winner = "p2"

ELSE

winner = "p1"

END IF

scoring()

END IF

update\_board(move, piece)

END execute

### Test Data

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Value | Expected Output | Reason for inclusion |
| Move | [(2,2), (2, 4)] | Board updates, score increments (by 0) | To test simple base case |
| Piece | “wp” |
| Attacking | “” |
| Move | [(7,6), (5,4)] | Scores are submitted, game is over | To test a game ending move |
| Piece | “bb” |
| Attacking | “wk” |

### Arguments to be Passed

* execute takes 3 arguments
  + move: a 2D array or integers [from(x, y), to(x, y)]
  + piece: a string of length 2 first letter = colour, second letter = piece
  + attacking: a string that stores the piece, if any, that is being attacked
* execute outputs the move onto the board and submits the writes the scores into a file

## update\_score

### Data Dictionary

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| Attacking | String | Local | SS | Stores the acronym of the piece that is being attacked, if there is no piece being attacked THEN it is "" |
| b | Integer | local constant | X | Stores the value of taking a bishop (3) |
| h | Integer | local constant | X | Stores the value of taking a horse (3) |
| k | Integer | local constant | X | Stores the value of taking a king (0) |
| p | Integer | local constant | X | Stores the value of taking a pawn (1) |
| q | Integer | local constant | X | Stores the value of taking a queen (9) |
| r | Integer | local constant | X | Stores the value of taking a rook (5) |

### Pseudocode

BEGIN update\_score(attacking)

REM will increase the score appropriate to the piece taken and the player taking it

const p = 1 REM the scores of taking each piece

const r = 5

const h = 3

const b = 3

const q = 9

const k = 0

IF turn = "p1" THEN REM will increment p1's score

CASEWHERE Right(attacking)

CASE "p" : p1score = p1score + p

CASE "r" : p1score = p1score + r

CASE "h" : p1score = p1score + h

CASE "b" : p1score = p1score + b

CASE "q" : p1score = p1score + q

CASE "k" : p1score = p1score + k

END CASE

ELSE

CASEWHERE Right(attacking) REM increment p2’s Score

CASE "p" : p2score = p2score + p

CASE "r" : p2score = p2score + r

CASE "h" : p2score = p2score + h

CASE "b" : p2score = p2score + b

CASE "q" : p2score = p2score + q

CASE "k" : p2score = p2score + k

END CASE

END IF

END update\_score

### Test Data

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Value | Expected Output | Reason for Inclusion |
| Attacking | “bq” | Increment score by 10 | To test a simple case |
| Attacking | “wb” | Increment score by 3 | To test both colours |
| Attacking | “wk” | Increment score by 0 | Test that it works with king |

### Arguments to be Passed

* update\_score takes one argument
  + attacking: a string that stores the piece, if any, that is being attacked
* update\_score returns an updated score of the player who is attacking

## update\_board

### Data Dictionary

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| Move | Array (integer) | Local | (2, 2) | Stores the x, y co-ordinates of the from and to variables for each move |
| Piece | String | Local | SS | Stores the acronym of the piece being moved |

### Pseudocode

BEGIN update\_board(move, piece)

picbox(move(1,1), move(1,2)).image = Nothing

picbox(move(2,1), move(2,2)).image = "filepath\" + piece + ".jpg"

board(move(1,1), move(1,2)) = ""

board(move(2,1), move(2,2)) = piece

END update\_board

### Test Data

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Value | Expected Output | Reason for Inclusion |
| Move | [(3,5), (4,4)] | Move a black king and update the board | To test a basic board update |
| Piece | “bk” |
| Move | [(3,2), (3,4)] | Move white rook and update board | To test a white piece |
| Piece | “wr” |

### Arguments to be Passed

* update\_board takes two arguments
  + move: a 2D array or integers [from(x, y), to(x, y)]
  + piece: a string of length 2 first letter = colour, second letter = piece
* update\_board returns an updated board array and board

## scoring

### Data Dictionary

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| scores | array (integer) | local | (variable length) | Stores the scores of previous players of the game |

### Pseudocode

BEGIN scoring()

scores = () REM an array of integers

get\_high\_scores(scores)

scores = sort\_scores(scores)

disp\_high\_scores(scores)

write\_file(p1score)

write\_file(p2score)

END scoring

### Test Data

Scoring takes no arguments and cannot be tested without get\_high\_scores, sort\_scores, disp\_high\_scores and write file all being fully tested or having stubs placed in all of them.

### Arguments to be Passed

* scoring takes no arguments
* scoring reads highscore.txt file, outputs the sorted scores and THEN appends the new score to the file

## get\_high\_scores

### Data Dictionary

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| counter | integer | local | X | a counter used to count the length of the file and read it into an array |
| scores | array (integer) | local | (variable length) | Stores the scores of previous players of the game |

### Pseudocode

BEGIN get\_high\_scores(ByRef scores)

open scores.txt for input

counter = 1

WHILE Not EOF

scores(counter) = file.readline

counter = counter + 1

ReDim Preserve scores(counter)

REM makes scores 1 element longer while preserving the contents

END WHILE

file.close() REM closes the file

REM scores now contains all the list of scores

END get\_high\_score

### Test Data

get\_high\_scores takes the empty array scores and reads a highscores file into it. To test this sub, you could place some test data into the file, call get\_high\_scores and THEN output the array scores into a list box

BEGIN test on click

REM designed to test get\_high\_scores()

get\_high\_scores()

FOR i = 1 to len(scores)

Output scores(i) into list box

NEXT i

END test

### Arguments to be Passed

* get\_high\_scores takes no arguments
* get\_high\_scores returns scores array filled with scores

## write\_file

### Data Dictionary

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| Score | Integer | local | XXXX | Is the score that is to be written to the file |

### Pseudocode

BEGIN write\_file(score)

REM writes a given score to the scores file

open file for writing

file.append(score)

REM writes the score to the bottom of the file

file.close

END write\_file

### Test Data

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Value | Expected Output | Reason for Inculsion |
| Score | 18 | A file with 18 as the last score | To test a basic case |
| Score | 0 | A file with 0 as the last score | To test a boundary case |

### Arguments to be Passed

* write\_file takes one argument
  + score: an integer that represents a score to be written to ta file
* write\_file returns a file with score appended to the end

## disp\_high\_scores

### Data Dictionary

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| length | integer | local | X | the length of the array being processed |
| Scores | array (integer) | local | (variable length) | Stores the scores of previous players of the game |

### Pseudocode

BEGIN disp\_high\_scores(scores)

length = len(scores)

FOR i = 1 to length

output scores(i) REM into a list

NEXT i

END disp\_high\_scores

### Test Data

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Value | Expected Output | Reason for Inclusion |
| Scores | (44,34,32,21,11,10,6) | A list with the contents of scores in it | To test a standard case of descending numbers |
| Scores | (34,74,42,12,2,0) | A list with the contents of scores in it | To test an unordered list |

### Arguments to be Passed

* disp\_high\_scores takes one argument
  + scores: an array filled with scores sorted from largest to smallest
* disp\_high\_scores outputs a list of the sorted scores

# Functions

## check\_pawn

### Data Dictionary

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| a | Boolean | local | True, False | Used to check if the move satisfies one of the four cases for a valid move |
| attacking | boolean | local | True, False | Stores value indicating if the piece is in its starting position |
| b | boolean | local | True, False | Used to check if the move satisfies one of the four cases for a valid move |
| c | boolean | local | True, False | Used to check if the move satisfies one of the four cases for a valid move |
| change\_x | integer | local | X | Stores the value of the change in the x component of the pieces move |
| change\_y | integer | local | X | Stores the value of the change in the y component of the pieces move |
| d | boolean | local | True, False | Used to check if the move satisfies one of the four cases for a valid move |
| move | Array (integer) | local | (2, 2) | Stores the x, y co-ordinates of the from and to variables for each move |
| piece | string | local | SS | Stores the acronym of the piece being moved |
| starting | boolean | local | True, False | Stores value indicating if the piece is in its starting position |
| taken\_piece | string | local | SS | Stores the value of any pieces taken, could either be "" or a 2 letter acronym of the piece. E.g. "bh" |

### Pseudocode

BEGIN FUNCTION check\_pawn(piece, move)

REM requires access to the board array

taken\_piece = board(move(2,1),move(2,2))

IF Left(piece) = Left(taken\_piece) THEN

REM will check that the piece that is to be taken by the rook is REM not of the same colour as it

Return False

END IF

change\_x = abs(move(1,1) - move(2,1))

REM the change in the x component of the pawn

REM the abs is used because it does not matter if the pawn moves 1 REM space Left or Right

change\_y = move(2,2) - move(1,2)

REM the change in the y component of the pawn

attacking = False

starting = False

IF board(move(2, 1), move(2, 2)) <> "" THEN

REM check where the pawn is moving too

attacking = True

END IF

IF move(1,2) = 2 and Left(piece) = "W" or move(1,2) = 7 and Left(piece) = "B" THEN

REM gets the y component of the starting position and checks if REM the pawn is starting

starting = True

END IF

REM these are the four boolean cases that each pawn could be in

a = attacking and Not starting and ((change\_y = 1 and Left(piece) = "W") or (change\_y = -1 and Left(piece) = "B")) and change\_x = 1

REM case where pawn is attacking

b = Not attacking and starting and change\_x = 0 and (((change\_y = 1 or change\_y = 2) and Left(piece) = "W") or ((change\_y = -1 or change\_y = -2) and Left(piece) = "B"))

REM case where the pawn is moving 1 or 2 spaces forward from REM starting

c = Not attacking and Not starting and change\_x = 0 and ((change\_y = 1 and

Left(piece) = "W") or (change\_y = -1 and Left(piece) = "B"))

REM case where the pawn is moving 1 space forward, not attacking,

REM not starting

d = attacking and starting and change\_x = 1 and ((change\_y = 1 and Left(piece) = "W") or (change\_y = -1 and Left(piece) = "B")

REM case where the pawn is attacking on the first move

IF a or b or c or d THEN

Return True

END IF

Return False

END FUNCTION check\_pawn

### Test Data

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Value | Expected Output | Reason for Inclusion |
| Move | [(4, 7), (4,5)] | True | To test a valid, starting move (2 spaces forward) |
| Piece | “bp” |
| Move | [(4,7), (4,6)] | True | To test a valid starting move (1 space forward) |
| Piece | “bp” |
| Move | [(6,4), (6,6)] | False | To test an invalid non-starting move (2 spaces forward) |
| Piece | “wp” |
| Move | [(3, 3), (4,4)] | True | To test a valid attacking move |
| Piece | “wp” |
| Attacking (local) | True |
| Move | [(3,3), (4,4)] | False | To test an invalid attacking move |
| Piece | “wp” |
| Attacking (local) | False |

Note: attacking is based on the global board array and is found within the function

### Arguments to be Passed

* check\_pawn takes two arguments
  + move: a 2D array or integers [from(x, y), to(x, y)]
  + piece: a string of length 2 first letter = colour, second letter = piece
* check\_pawn returns a Boolean value indicating whether a move is valid

## check\_rook

### Data Dictionary

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| change\_x | integer | local | X | Stores the value of the change in the x component of the pieces move |
| change\_y | integer | local | X | Stores the value of the change in the y component of the pieces move |
| Move | Array (integer) | local | (2, 2) | Stores the x, y co-ordinates of the from and to variables for each move |
| Piece | string | local | SS | Stores the acronym of the piece being moved |
| taken\_piece | string | local | SS | Stores the value of any pieces taken, could either be "" or a 2 letter acronym of the piece. E.g. "bh" |
| X | integer | local | X | Stores the current x co-ordinate of the cell being checked (used in a for next loop) |
| Y | integer | local | X | Stores the current y co-ordinate of the cell being checked (used in a for next loop) |

### Pseudocode

BEGIN FUNCTION check\_rook(piece, move)

REM requires access to the board array

REM rooks can move vertically and horizontally for any distance

taken\_piece = board(move(2,1),move(2,2))

IF Left(piece) = Left(taken\_piece) THEN

REM will check that the piece that is to be taken by the rook is REM not of the same colour as it

Return False

END IF

change\_x = move(2,1) - move(1,1)

REM a negative value indicates moving Left

change\_y = move(2,2) - move(1,2)

REM a negative value indicates moving backwards

IF change\_x <> 0 and change\_y <> 0 THEN

REM checks that the move is not diagonal

Return False

END IF

REM checks movement in the x co-ordinate

IF change\_x > 0 THEN

REM checks wether there are any pieces in between the rook and its REM destination

x = move(1,1) + 1

REM the +1 is so that the FOR...NEXT loop does not check the tile REM its self

FOR i = x to x + change\_x – 1

REM this will cycle through the board until it reaches the

REM specified end point

IF board(move(1,2),i) <> "" THEN

Return False

END IF

NEXT i

END IF

IF change\_x < 0 THEN

REM case: moving backwards

x = move(1,1) – 1

REM the -1 is so that the FOR...NEXT loop does not check the tile

REM it’s self

FOR i = x to x + change\_x + 1 Step -1

REM cycles from Right to Left through the board

IF board(i, move(1,2)) <> ""

Return False

END IF

NEXT i

END IF

REM checks movement in the y co-ordinate

IF change\_y > 0 THEN

y = move(1,2) + 1

FOR i = y to y + change\_y - 1

IF board(move(1,1),i) <> "" THEN

Return False

END IF

NEXT i

END IF

IF change\_y < 0 THEN

y = move(1,2) - 1

FOR i = y to y + change\_y + 1

IF board(move(1,1),i) <> "" THEN

Return False

END IF

NEXT i

END IF

END FUNCTION check\_rook

### Test Data

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Value | Expected Output | Reason for Inclusion |
| Move | [(1,1), (1,5)] | True | To test a valid vertical move |
| Piece | “wr” |
| Move | [(4,4), (2,4)] | True | To test a valild horizontal move |
| Piece | “br” |
| Move | [(4,4), (6,6)] | False | to test an invalid move |
| Piece | “wr” |

### Arguments to be Passed

* check\_rook takes two arguments
  + move: a 2D array or integers [from(x, y), to(x, y)]
  + piece: a string of length 2 first letter = colour, second letter = piece
* check\_rook returns a Boolean value indicating whether a move is valid

## check\_horse

### Data Dictionary

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| A | boolean | local | True, False | Stores a boolean calculation relating to the change in x and y co-ordinates |
| B | boolean | local | True, False | Stores a boolean calculation relating to the change in x and y co-ordinates |
| change\_x | integer | local | X | Stores the value of the change in the x component of the pieces move |
| change\_y | integer | local | X | Stores the value of the change in the y component of the pieces move |
| Move | Array (integer) | local | (2, 2) | Stores the x, y co-ordinates of the from and to variables for each move |
| Piece | string | local | SS | Stores the acronym of the piece being moved |
| taken\_piece | string | local | SS | Stores the value of any pieces taken, could either be "" or a 2 letter acronym of the piece. E.g. "bh" |

### Pseudocode

BEGIN FUNCTION check\_horse(piece, move)

REM for horses move see documentation

taken\_piece = board(move(2,1),move(2,2))

IF Left(piece) = Left(taken\_piece) THEN

REM will check that the piece that is to be taken by the rook is

REM not of the same colour as it

Return False

END IF

change\_x = abs(move(2,1) - move(1,1))

REM abs because +/- x becomes x

change\_y = abs(move(2,2) - move(1,2))

a = change\_x = 1 and change\_y = 2

b = change\_x = 2 and change\_y = 1

IF a or b THEN

Return True

ELSE

Return False

END IF

END FUNCTION check\_horse

### Test Data

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Value | Expected Output | Reason for Inclusion |
| Move | [(2,2), (3,4)] | True | To test a valid horse move |
| Piece | “wh” |
| Move | [(2,2), (2,4)] | False | To test an invalid horse move |
| Piece | “wh” |
| Move | [(5,5), (7,6)] | True | To test a horizontal horse move |
| Piece | “bh” |

### Arguments to be Passed

* check\_horse takes two arguments
  + move: a 2D array or integers [from(x, y), to(x, y)]
  + piece: a string of length 2 first letter = colour, second letter = piece
* check\_horse returns a Boolean value indicating whether the move is valid

## check\_bishop

### Data Dictionary

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| Move | Array (integer) | Local | (2, 2) | Stores the x, y co-ordinates of the from and to variables for each move |
| Piece | string | Local | SS | Stores the acronym of the piece being moved |
| taken\_piece | string | Local | SS | Stores the value of any pieces taken, could either be "" or a 2 letter acronym of the piece. E.g. "bh" |
| Temp | string | Local | SS | Stores a string of 2 different signs that tell the computer wether to increment of decrement. e.g. "+-" or "--" |
| X | integer | Local | X | Stores the x co-ordinate of a tile location that is being checked. |
| Y | integer | Local | X | Stores the y co-ordinate of a tile location that is being checked. |

### Pseudocode

BEGIN FUNCTION check\_bishop(piece, move)

REM bishops can move diagonally FOR any distance

taken\_piece = board(move(2,1),move(2,2))

IF Left(piece) = Left(taken\_piece) THEN

REM will check that the piece that is to be taken by the rook is REM not of the same colour as it

Return False

END IF

change\_x = move(2,1) - move(1,1)

change\_y = move(2,2) - move(1,2)

IF abs(change\_x) <> abs(change\_y) THEN

REM this checks that the move is diagonal

Return False

END IF

temp = get\_increment(change\_x, change\_y)

x = move(1,1)

y = move(1,2)

REM will cycle through the tiles in between the piece and its

REM destination

WHILE x < move(2,1) - 1 or x > move(2,1) + 1

REM uses the fact that abs(x) = abs(y)

IF Left(temp) = "+" THEN

x = x + 1

ELSE

x = x - 1

END IF

IF Right(temp) = "+" THEN

y = y + 1

ELSE

y = y - 1

END IF

IF board(x,y) <> "" and x <> move(2,1) THEN

REM checks that the piece is not "jumping" any other pieces

Return False

END IF

END WHILE

END FUNCTION check\_bishop

### Test Data

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Value | Expected Output | Reason for Inclusion |
| Move | [(3,4), (5,6)] | True | To test a valid bishop move (forward) |
| Piece | “bb” |
| Move | [(2,4), (4, 2)] | True | To test a valid bishop move (backward) |
| Piece | “wb” |
| Move | [(3, 4), (6,4)] | False | To test n invalid bishop move |
| Piece | “bb” |
| Move | [(1,1), (8,8)] | True | To test a boundary case |
| Piece | “wb” |

### Arguments to be Passed

* check\_bishop takes two arguments
  + move: a 2D array or integers [from(x, y), to(x, y)]
  + piece: a string of length 2 first letter = colour, second letter = piece
* check\_bishop returns a Boolean value that indicates whether a move is valid

## get\_increment

### Data Dictionary

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| change\_x | integer | Local | X | Stores the value of the change in the x component of the pieces move |
| change\_y | integer | Local | X | Stores the value of the change in the y component of the pieces move |

### Pseudocode

BEGIN FUNCTION get\_increment(change\_x, change\_y)

REM used with the check\_bishop function

REM this checks wether to increment or decrement x and y and

REM Returns a string

IF change\_x > 0 THEN

REM if change\_x > 0, then the piece is moving diagonally to the REM Right

IF change\_y > 0 THEN

REM if change\_y > 0 THEN the piece is moving diagonally forward

Return '++'

ELSE

Return '+-'

END IF

ELSE

IF change\_y > 0 THEN

Return '-+'

ELSE

Return '--'

END IF

END IF

END FUNCTION get\_increment

### Test Data

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Value | Expected Output | Reason for Inclusion |
| Change\_x | 2 | “++” | To test a “++” case |
| Change\_y | 2 |
| Change\_x | 4 | “+-“ | To test a “+-“ case |
| Change\_y | -3 |
| Change\_x | -2 | “- -“ | To test a “- -“ case |
| Change\_y | -5 |
| Change\_x | -5 | “-+” | To test a “-+” case |
| Change\_y | 2 |

### Arguments to be Passed

* get\_increment takes to arguments
  + change\_x: an integer representing the change in the x vector of a move
  + change\_y: an integer representing the change in the y-vector of a move
* get\_increment returns a string of length 2 that indicates how to increment the x, y variables

## check\_queen

### Data Dictionary

|  | | | | |
| --- | --- | --- | --- | --- |
| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| change\_x | integer | Local | X | Stores the value of the change in the x component of the pieces move |
| change\_y | integer | Local | X | Stores the value of the change in the y component of the pieces move |
| Move | Array (integer) | Local | (2, 2) | Stores the x, y co-ordinates of the from and to variables for each move |
| Piece | string | Local | SS | Stores the acronym of the piece being moved |
| taken\_piece | string | Local | SS | Stores the value of any pieces taken, could either be "" or a 2 letter acronym of the piece. E.g. "bh" |

### Pseudocode

BEGIN FUNCTION check\_queen(piece, move)

REM queen can move vertically, horizontally or diagonally for any REM distance

taken\_piece = board(move(2,1),move(2,2))

IF Left(piece) = Left(taken\_piece) THEN

REM will check that the piece that is to be taken by the rook is REM not of the same colour as it

Return False

END IF

change\_x = move(2,1) - move(1,1)

change\_y = move(2,2) - move(1,2)

IF abs(change\_x) = abs(change\_y) THEN

Return check\_bishop(piece, move)

END IF

IF (change\_x = 0 and change\_y <> 0) or (change\_x <> 0 and change\_y = 0) THEN

Return check\_rook(piece, move)

END IF

Return False

END FUNCTION check\_queen

### Test Data

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Value | Expected Output | Reason for Inclusion |
| Move | [(4,4), (4,2)] | True | To test a backwards move |
| Piece | “bq” |
| Move | [(4,4), (6,2)] | True | To test a back-right diagonal move |
| Piece | “wq” |
| Move | [(4,4), (6,4)] | True | To test a left move |
| Piece | “bq” |
| Move | [(4,4), (6,6)] | True | To test a forward-right diagonal move |
| Piece | “wq” |
| Move | [(4,4), (4,6)] | True | To test forward move |
| Piece | “bq” |
| Move | [(4,4), (2,6)] | True | To test a forward-left diagonal move |
| Piece | “wq” |
| Move | [(4,4), (2,4)] | True | To test a left move |
| Piece | “bq” |
| Move | [(4,4), (2,2)] | True | To test a backwards-left diagonal move |
| Piece | “wq” |
| Move | [(4,4), (7,3)] | False | To test an invalid move |
| Piece | “bq” |

### Arguments to be Passed

* check\_queen takes two arguments
  + move: a 2D array or integers [from(x, y), to(x, y)]
  + piece: a string of length 2 first letter = colour, second letter = piece
* check\_queen returns a Boolean value that indicates whether a move is valid

## check\_king

### Data Dictionary

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| change\_x | integer | local | X | Stores the value of the change in the x component of the pieces move |
| change\_y | integer | local | X | Stores the value of the change in the y component of the pieces move |
| move | array (integer) | local | (2, 2) | Stores the x, y co-ordinates of the from and to variables for each move |
| piece | string | local | SS | Stores the acronym of the piece being moved |
| taken\_piece | string | local | SS | Stores the value of any pieces taken, could either be "" or a 2 letter acronym of the piece. E.g. "bh" |

### Pseudocode

BEGIN FUNCTION check\_king(piece, move)

REM king can move vertically, horizontally or diagonally 1 space

taken\_piece = board(move(2,1),move(2,2))

IF Left(piece) = Left(taken\_piece) THEN

REM will check that the piece that is to be taken by the rook is REM not of the same colour as it

Return False

END IF

change\_x = move(2,1) - move(1,1)

change\_y = move(2,2) - move(1,2)

IF (change\_x => -1 and change\_x <= 1) and (change\_y => -1 and change\_y <= 1) THEN

Return True

END IF

Return False

END FUNCTION check\_king

### Test Data

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Value | Expected Output | Reason for Inclusion |
| Move | [(4,4), (4,3)] | True | To test a backwards move |
| Piece | “bk” |
| Move | [(4,4), (5,3)] | True | To test a back-right diagonal move |
| Piece | “wk” |
| Move | [(4,4), (5,4)] | True | To test a left move |
| Piece | “bk” |
| Move | [(4,4), (5,5)] | True | To test a forward-right diagonal move |
| Piece | “wk” |
| Move | [(4,4), (4,5)] | True | To test forward move |
| Piece | “bk” |
| Move | [(4,4), (3,5)] | True | To test a forward-left diagonal move |
| Piece | “wk” |
| Move | [(4,4), (3,4)] | True | To test a left move |
| Piece | “bk” |
| Move | [(4,4), (3,3)] | True | To test a backwards-left diagonal move |
| Piece | “wk” |
| Move | [(4,4), (7,3)] | False | To test an invalid move |
| Piece | “bk” |

### Arguments to be Passed

* check\_king takes two arguments
  + move: a 2D array or integers [from(x, y), to(x, y)]
  + piece: a string of length 2 first letter = colour, second letter = piece
* check\_king returns a Boolean value that indicates whether a move is valid

## sort\_scores

### Data Dictionary

| **Variable Name** | **Data Type** | **Scope** | **Format** | **Description** |
| --- | --- | --- | --- | --- |
| end\_unsorted | integer | local | X | the length of the unprocessed part of the array |
| I | integer | local | X | counter used to sort the array |
| Max | integer | local | X | stores the largest element in the unsorted part of the array |
| posmax | integer | local | X | stores the position of the maximum element |
| Array | array (integer) | local | (variable length) | A list of integers that will be the scores of the |
| Temp | integer | local | X | used to swap two array elements |

### Pseudocode

BEGIN FUNCTION sort\_scores(array) REM selection sort

REM through much testing in python, selection sort has proven to REM be quite an efficient method of sorting large amounts of data REM compared to bubble sort and insertion sort

REM the development of a quicksort is underway to make this

REM process even faster

end\_unsorted = len(array)

temp = 0

WHILE end\_unsorted > 1

i = 1

min = array(i)

posmin = i

WHILE i < end\_unsorted

i = i + 1

IF array(i) < min THEN

min = array(i)

posmin = i

END IF

END WHILE

temp = array(posmin)

array(posmin) = array(end\_unsorted)

array(end\_unsorted) = temp

end\_unsorted = end\_unsorted - 1

END WHILE

Return array

END FUNCTION sort\_scores

### Test Data

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Value | Expected Output | Reason for Inclusion |
| Array | (1,2,3,4,5,6,7) | (7,6,5,4,3,2,1) | To test a sorted ascending array |
| Array | (7,6,5,4,3,2,1) | (7,6,5,4,3,2,1) | To test a sorted descending array |
| Array | (3,5,1,6,8,3,5) | (8,6,5,5,3,3,1) | To test an unsorted array |

### Arguments to be Passed

* sort\_scores takes one argument
  + array: and array of integers
* sort\_scores returns a descending sorted array

# Data Structures

### Arrays

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Array Name | Dimension | Index | Data Type | Scope |
| Move | 2 | 1 to 2, 1 to 2 | Integer | Global |
| picbox\_array | 2 | 1 to 8, 1 to 8 | Picture box | Global |
| Board | 2 | 1 to 8, 1 to 8 | String | Global |
| Scores | 1 | Variable | Integer | Global |

# Files

File Name: “Scores.txt”

File Location: “C:\Users\Andrew\Documents\Visual Studio 2012\Projects\chessgame\chessgame\bin\Debug”

Type Of File: Sequential

Structure of File: score1, score2 score3… No sentinel value. One Score per Line, ordered

Sample:

32

32

30

29

25

24