COSC1076 – Advanced Programming Techniques

Semester 2, 2014

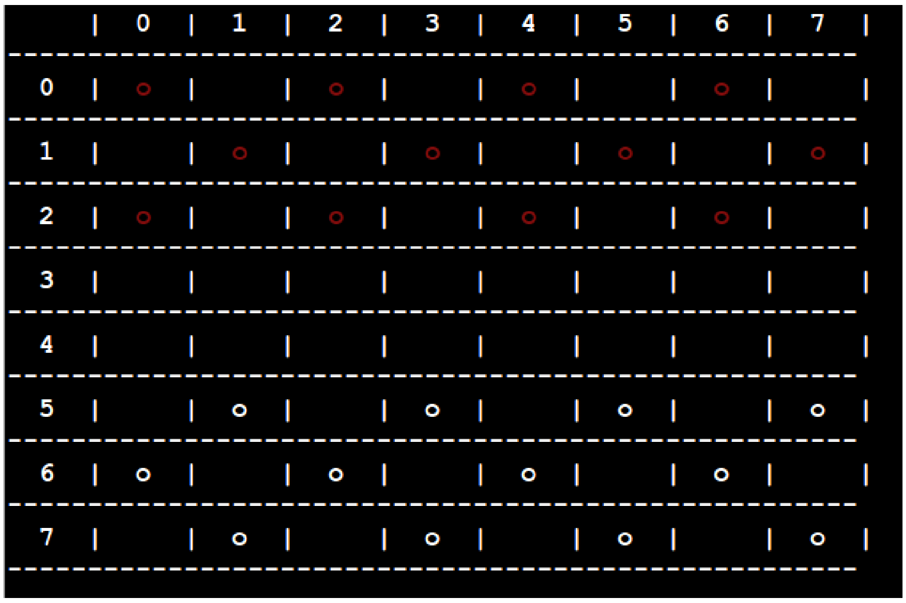
Assignment 1 – English Draughts.

In this assignment, you are asked to implement a moderate size program to demonstrate early knowledge of C programming principles.

The concepts covered include:

- Data types and basic operators.
- Branching and looping programming constructs.
- Basic input and output functions.
- Strings.
- Functions.
- Pointers.
- Arrays.
- Basic modularisation
- C Structs

We have decided this year to provide you what will be a fun assignment, a two player version of the board game, “English draughts”. There will be two players of this game, a red player and a white player both controlled via the keyboard. The game will start with the computer initialising the board to be blank and then laying out the tokens for both players. The tokens are laid out on the board to begin with as follows (red is always at the top):



```
    |  0  |  1  |  2  |  3  |  4  |  5  |  6  |  7  |
-----------------------------------------------------
 0  |  o  |     |  o  |     |  o  |     |  o  |     |
-----------------------------------------------------
 1  |     |  o  |     |  o  |     |  o  |     |  o  |
-----------------------------------------------------
 2  |  o  |     |  o  |     |  o  |     |  o  |     |
-----------------------------------------------------
 3  |     |     |     |     |     |     |     |     |
-----------------------------------------------------
 4  |     |     |     |     |     |     |     |     |
-----------------------------------------------------
 5  |     |  o  |     |  o  |     |  o  |     |  o  |
-----------------------------------------------------
 6  |  o  |     |  o  |     |  o  |     |  o  |     |
-----------------------------------------------------
 7  |     |  o  |     |  o  |     |  o  |     |  o  |
-----------------------------------------------------
```

Each player has 12 tokens placed in the locations specified. The token's location may be referred to by two digits in a row, separated by a comma (a tuple) – firstly the column and then the row. So the tuple "0,0" would refer to the top left cell and the tuple "7,0" to the top right, "0,7" to the bottom left, and so on.

A lower case "o" will be used for a normal token and the token for a king will up an upper case "O".

Before the game begins, the players will be given the opportunity to enter their names. Players will be randomly assigned a colour, red or white. The red player will always move first and then players will take turns with their moves until one player can no longer make a legal move.

Tokens can only move in a diagonal direction and normal tokens can only move towards the opposite side from where they started. So, normal white tokens can only move up the board and normal red tokens can only move down the board. A token becomes a king by reaching the opposite end of the board from where it started. Once it becomes a king, a token can move in either direction.

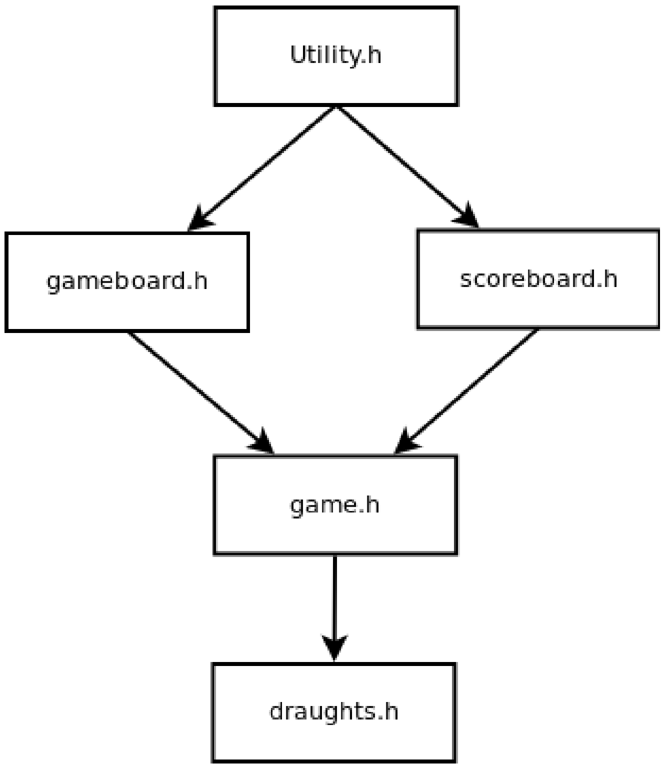
There are two valid moves that can be made – a normal move where a token can move one square at a time and an attack move which is where a token jumps over a diagonally adjacent enemy . You can only jump one token at a time but once you have made an attack move if there is another token that can also be jumped you can do that immediately – you don't have to wait until your next turn to do so. Any tokens that are jumped are removed from the game.

The rules of the game as you will program it are according to the rules of "English Draughts" at the following url: http://www.mastersgames.com/rules/draughts-rules.htm Any alternate rules you may have come across do not apply.

Please ensure that you have read the document "Assignment Specifications – General Requirements" as they form part of the assessment requirements for each assignment in this course.

Startup Code

To help shape your solution we have provided you with the following modules (.c and .h pairs). For your reference, please note the layout below which demonstrates the flow of the layout.



utility.h/c

Contains utility functions to perform generic actions. An example might be a get_int() function.

gameboard.h/c

Contains all the functions and declarations relating to the gameboard.

scoreboard.c/h

Contains all the functions and structs relating to the scoreboard.

game.h/c

Contains all the functions and declarations needed to control the logic and flow of the game.

draughts.c/h

Contains main and all the functions and declarations that manage the user interface (such as the menu).

Below we will outline the data structures we have provided to assist you in your solution to this assignment. **Please note that these data structures form part of the startup code and cannot be modified.** The startup code also includes function prototypes. You must use this startup code. You should feel free however to create additional variables, constants and functions to help you with your solution. In fact, it is impossible to get full marks without doing this.

Provided Data structures

Each location on the board can be described by an x-y pair of coordinates as entered by a player when they wish to move a token on the board. This will be represented in the startup code by the following struct:

```
struct location
{
     unsigned x,y;
};
```

A "move" is represented in the game by a pair of locations – a start and an end:

```
struct move
{
     struct location start;
     struct location end;
};
```

We are also providing you with some enumerations which should be helpful in having more meaningful code that is easier to maintain. An enumeration is a series of integer values that are related together. For example, each cell of the game board is defined by the following enumeration:

```
enum cell_contents
{
      EMPTY, RED, WHITE, K_RED, K_WHITE
};
```

In other words, each cell that makes up the game board could be empty, it could be holding a normal red or white token or it could be holding a king token of either red or white colour.

The game board itself is an array of these items:

```
enum cell_contents gameboard[BOARDWIDTH][BOARDHEIGHT];
```

There are two things you might notice about the above declaration – firstly, the type is not "cell_contents" but "enum cell_contents" – the keyword enum is part of the type name. This is true with structs as well. You are able to use the "typedef" language construct to rename types if you wish but the contents must remain the same. You should also feel free to add types of your own – in the startup code we have provided only those datatypes that we feel are essential to get the job done but they aren't the end of the story – in fact you should experiment a little with the language as that is how we learn.

We are also providing you with a global variable template gameboard. You cannot use this to play a game of draughts but you can copy the contents from here to set or reset your gameboard at the beginning of each game.

```
static const enum cell_contents master_board[BOARDWIDTH]
[BOARDHEIGHT] =
{
    {RED, EMPTY, RED, EMPTY, RED, EMPTY, RED, EMPTY},
    ...
    {EMPTY, WHITE, EMPTY, WHITE, EMPTY, WHITE, EMPTY, WHITE}
};
```

We also provide you with the constants for printing the coloured (red and white) tokens based on the value in the cell as follows.

```
#define WHITE_DISPLAY "\x1b[39m" "o"
#define WHITE_KING_DISPLAY "\x1b[39m" "O"
#define RED_DISPLAY "\x1b[31m" "o"
#define RED_KING_DISPLAY "\x1b[31m" "O"
#define WHITE_RESET "\x1b[39m"
```

We have provided the following enumeration which will be used to determine if a move is valid. The use of this will be explained under the section “Validate Winner” but essentially you will use this to determine whether a move is valid and what type of move it was.

```
enum move_type
{
    ATTACK, NORMAL, INVALID
};
```

We have provided a player struct (next page) which will contain basic information about a player, the name and the colour. This will make it easier to randomise the colour assignment and to pass information about players to different functions.

```
enum colour
{
     P_RED, P_WHITE
};
```

```
struct player
{
     char name[MAX_NAME_LEN+1];
     enum colour col;
};
```

We have also provided you a scoreboard where you should record the latest winners of the games played. We want you to store the names of the last ten winning players. You should also record the names of the losers of those games. To make this easier we have provided the following data structure:

```
struct result
{
     char won[MAX_NAME_LEN+1];
     char lost[MAX_NAME_LEN+1];
};
```

In main we have the following declaration of the array of ten results:

```
struct result scoreboard[SCOREBOARDSIZE];
```

Please see “Initialise Scoreboard” and “Update Scoreboard” for how you will use this in your program.

We have also provided you a definition of the BOOLEAN datatype. C does not have a BOOLEAN type actually declared although the value 0 evaluates to FALSE and all other values evaluate to TRUE. For the reasons of consistency, we provide you with this type:

```
typedef enum
{
     FALSE, TRUE
} BOOLEAN;
```

Functional Requirements (75 marks)

This section describes in detail all functionality requirements for assignment 1. A rough indication of the number of marks for each requirement is provided.

You are expected to ensure that your implementation mirrors the output shown in this specification, including (where applicable) formatting.

Requirement 1: Main menu – 5 marks

You must display an interactive menu containing the following options:

```
English Draughts - Main Menu
1) Play Game
2) Display Winners
3) Reset scoreboard
4) Quit
```

Note: when "Play Game" is selected, both players must be prompted for their names as you will need to pass that information into the play_game function.

Requirement 2: Initialise the Score Board (5 Marks)

At program startup, the scoreboard needs to be initialised so that all the player names are empty strings. This will ensure that we only print only those entries that represent a real game.

This is implemented using the function:

```
void scoreboard_init(struct result * scoreboard);
```

Requirement 3: Initialise The Game Board – (5 marks)

You need to initialise the game board by copying our global constant array master_board into the gameboard declared in the play_game function. This involves a simple copying of a 2 dimensional array using for loops. The provided global constant master_board (see startup code) is the only global variable that should exist in your program. The function you need to implement for this requirement is:

```
void init_gameboard(enum cell_contents board[][BOARDWIDTH]);
```

Requirement 4: Game Loop (5 Marks)

This is the function that is called from the main menu to play a game.

The game loop for this program is relatively simple. Your program should implement the following algorithm:

```
Randomly allocate colours to players
While not quitting and not end of game
     displayboard
     if opposing player has no valid moves
          stop while
     end if
     red_player_turn
     displayboard
     if opposing player has no valid moves
          stop while
     end if
     white_player_turn
End while
At the end of the game, return the name of the winner.
```

The function to implement this functionality is:

```
void play_game(char * human, char * computer,
     struct result * outcome);
```

Requirement 5: Player's Turn – 12 marks

A valid move in English Draughts is one of the following:

- A normal move: If the token is not a king, a diagonal move forward by one square, and if the token is a king a diagonal move one square in any direction.
- An attack move: Where the token jumps over an enemy's adjacent token into an empty square. This move could be done several times within one turn if there is the opportunity to do so.

With the above in mind, you will accept input from each player in turn, one move at a time as follows:

```
It is Jean Luc's turn. Please enter a move [press enter or
ctrl-D to quit the current game]: 0,2-1,3
```

This move would be a request to move the token currently at x=0,y=2 to x=1,y=3. This is an example of a normal move. If available, the user could also enter an attack move:

```
It is Jean Luc's turn. Please enter a move [press enter or
ctrl-D to quit the current game]: 0,2-2,4
```

If this move is a valid attack move (this moves jumps over one of the opposing player's tokens and removes it from the board) then the program should prompt the user to see if they want to make another attack move with the same token:

```
You attacked! Would you like to attempt a further attack move
with this token? [y for yes, n for no or press enter or ctrl-D
to quit the current game]: y

It is Jean Luc's turn. Please enter a move [press enter or
ctrl-D to quit the current game]: 2,4-4,6
```

The program should continue to prompt for and accept valid attack moves until the user elects to end their turn by entering an 'n' or by quitting.

Note that this differs from the rules given whereby a second attack move would be forced if such a move existed.

Note that if a user presses enter on an empty line or presses Ctrl-D then they should be returned to the menu. No result should be added to the scoreboard.

Note that a player cannot end a sequence of attack moves with a normal move.

A subsequent move in a sequence of attack moves is only valid if it relates to the same token used in the original move of the sequence. For example, if the starting move in a sequence was 0,2-2,4 then a subsequent move of 0,0-2,2 would invalid even though it might otherwise be a legal attack move.

If a player enters an invalid move you will need to display a clear error message detailing the nature of the error and reprompt.

You will need to parse each move entered by the user into pairs of locations (using the struct move type explained in the "Provided Datastructures" section above) to pass into the is_valid_move() function mentioned below.

Note: If you process input one character at a time you will not get full marks. You should use standard C functions such as strtok and implement full input validation.

At the end of a valid move, if the token is a normal token you should check if it has reached the other end of the board and if it has, make it a king.

Requirement 6: Validate Move – 12 marks

When a move is entered it will need to be validated as being a legal move within the game. This is where the struct move defined earlier in the startup code comes in. The function you need to implement for this is:

```
enum move_type is_valid_move(struct move next_move,
          struct player * current,
          enum cell_contents board[][BOARDWIDTH])
```

Requirement 7: Test for Winner (12 marks)

According to the rules that we are using, a game is won if the next player can make no valid moves. Therefore at the end of each players turn, the program should check whether the next player has any valid moves available.

The simplest algorithm would be to iterate over the two-dimensional board array as below and if no token belonging to the opposing player is found that could make a valid move, the game is over. An example algorithm is given below. You should feel free to use a more efficient algorithm if you wish.

```
For each column on the game board
     For each row on the game board
          If opposing token exists
               For each possible move (attack or normal)
                    If move is valid
                         return FALSE
               end for
          end if
     end for
end for
```

To do this you must implement and use the functions:

```
BOOLEAN test_for_winner(struct player * opposing,
     enum cell_contents board[][BOARDWIDTH]);
```

Note: This function can be complex and we suggest you leave it until you have finished all the other requirements. However you will need to have some method of ending a game in order to test the rest of your program. We suggest you employ an interim solution using the algorithm below so that you are able to test your game loop. Please note that if you submit your assignment with the following algorithm then you will receive NO MARKS for this section.

The following algorithm suggests an “interim solution” for test_for_winner(). This function will return true (the next player has moves available) roughly 70% of the time.

```
Generate a random number between 0 and 10
     if number is greater than 7
          return false
     else
```

```
            return true
```

Requirement 8: End The Game (2 Marks)

At the end of a game, the player should be asked if they wish to play again and if they do, a new game should be started and if they don't, return to the main menu. In either case, add_to_scoreboard() should be called to add the latest result to the scoreboard.

If the player selects that they wish to play again, then your program should assume that it is the same player and not re-prompt for name.

Requirement 9: Add Players to Scoreboard (5 Marks)

At the end of a game, the players who just played should be added to scoreboard[0] and all other results shifted down by one place. If the game is quit then no result is added to the scoreboard. In order to implement this option, the function:

```
void add_to_scoreboard(struct result * scoreboard,
     struct result * latest_game);
```

must be implemented and called.

Requirement 10: Display Score Board (5 marks)

You should display the scoreboard in the following fashion. You will need to leave enough space in your format for the longest name allowed by MAX_NAME_LEN. Please note that you only need to print out results until you reach the first empty string :

```
APT English Draughts Tournament - History of Games Played.
==========================================================
| Winner               | Loser                |
| ---------------------| ---------------------|
| Fred                 | Computer             |
| Computer             | Fred                 |
| Dave                 | Computer             |
```

Requirement 11: Reset Scores (2 Marks)

Resets all player slots to empty To implement this option you must implement and call the function:

```
    void reset_scoreboard(struct result * scoreboard);
```

Requirement 12: Quit (2 Marks)

Quits the program.

Requirement 13: Demonstration (5 marks)

A demonstration is required for Assignment #1 to show your progress. It will occur in your scheduled lab classes. The demonstration will be brief, and you must be ready to demonstrate in a two minute period when it is your turn. Buffer handling and input validation will not be assessed, so you are advised to incorporate these only after you have implemented the demonstration requirements. Coding conventions and practices too will not be scrutinised, but it is recommended that you adhere to such requirements at all times.

This demonstration will occur in the week beginning 18 August 2014. You will need to demonstrate the following, without, at this stage, requiring to implement validation. You must demonstrate that you have implemented:

Ability to compile/execute your program from scratch (1 mark).

Requirements 1, 3,5 and 6 implemented and running (1 mark each). That is, for full marks your program should display a main menu, initialise the gameboard and allow the user to start a game. The first player should be able to take at least one turn.

Submit your work (see Submission Instructions below) immediately after this demonstration (1 mark).

General requirements – (25 marks)

Please pay attention to the "Functional Abstraction", "Buffer Handling", "Input Validation" and "Coding conventions/practices" sections of the "Assignment Specifications General Requirements" document. These requirements are going to be weighted at 7, 5, 5 and 8 marks, respectively.

Submission Information

Submission date/time:

Submission details for Assignment 1 are as follows. Note that late submissions attract a marking deduction of 10% of the maximum mark attainable per day for the first 5 university days. After this time, a 100% deduction is applied.

CATEGORY	DUE DATE/TIME			PENALTY
On time	**Friday**	**29/08/2014**	**9:00pm**	**N/A**
1 day late	Thursday	30/08/2014	9:00pm	10% of total available marks
2 days late	Friday	31/08/2014	9:00pm	20% of total available marks
3 days late	Saturday	01/09/2014	9:00pm	30% of total available marks
4 days late	Sunday	02/09/2014	9:00pm	40% of total available marks
5 days late	Monday	03/09/2014	9:00pm	50% of total available marks
6 or more days late	Not accepted			100% of total available marks

We recommend that you avoid submitting late where possible because it is difficult to make up the marks lost due to late submissions.

Submission content:

For Assignment #1, you need to submit the following files:

draughts.c, draughts.h, game.c, game.h, scoreboard.c, scoreboard.h, gameboard.c, gameboard.h, utility.c, utility.h and readme.txt

readme.txt: you should include in this file any comments you have on this project and any messages for your markers.

NOTE: Please submit only the files listed above, marks will be deducted for submitting whole directories, and for submitting compiled binaries.

Please note that your program will need to compile cleanly (no warnings and no errors) using gcc v4.8.1 on the Coreteaching servers using the recommended flags: **-Wall -ansi -pedantic**

Submission instructions:

Create a zip archive using the following command on saturn or jupiter:`zip $(USER) *.c *.h *.txt`

This will create a .zip file with your username on the server, eg: if your student id is 3333333 then the file will be called: s3333333.zip.

Submit this archive to **Weblearn**, which is accessible via the learning hub.

[END OF ASSIGNMENT #1 INFORMATION]