# COSC1076/2207 Advanced Programming Techniques

## *Semester 2, 2015*

## *Assignment#1 – Connect 4*

## *Due Date: Friday 28th August, 9 pm.*

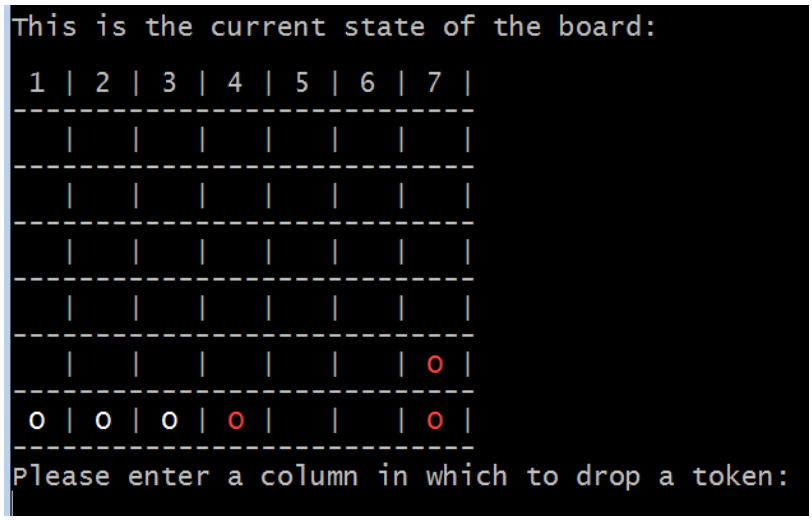
In this assignment, you are asked to implement a moderate size program to demonstrate early knowledge of C programming principles.

The concepts covered include:

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

We have decided this year to provide you what will be a fun assignment, a two player version of the game "Connect 4". Connect 4 is a rather simple game. Players take it in turns to place a token in a column on a board – the placed token drops down that column to the last available cell for it. Each player attempts to get four of their tokens in a row while blocking the opposing player for achieving the same objective.

The gameboard for this game will look as follows:



Before the game begins, the human player will be given the opportunity to enter their name. Each player will then be allocated a colour randomly – either red or white. The player allocated the white colour will go first and each player will be given the opportunity to drop a token onto the board until either player has 4 tokens in a row or there are no more possible moves.

**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. Also, please read the programmer documentation that we have uploaded to http://titan.csit.rmit.edu.au/~e70949/apt_a1_2015/ – this is the reference documentation for this program and any updates or clarifications made on blackboard will be reflected here.**

## Startup Code

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.

We are providing you with some enumerations and structs 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 board is defined by the following enumeration:

```
/**
 * defines the possible contents of each cell of the board
**/
enum cell_contents
{
     /** the cell does not contain a token **/
     C_EMPTY,
     /** the cell contains a red token **/
     C_RED,
     /** the cell contains a white token **/
     C_WHITE
};
```

In other words, each cell that makes up the game board could be empty, it could be holding a red or white counter.

Another enumeration that we provide you with defines a "boolean" datatype. C (certainly ansi C) does not have a built-in boolean datatype (that is, a datatype for true and false), and because of that, if we are going to have a boolean type, we will have to define it ourselves. The header file "bool.h" contains the following definition:

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

This definition contains a “tag name” (truefalse) that is used to define the enumeration. This is then wrapped in a typedef statement. A typedef statement renames one type as another. Here, we rename the “enum truefalse” to be the type BOOLEAN, which we can then use whenever we need a true/false type value. One other thing to take notice of is how the counting of values occurs. Remember that an enumeration is a set of related integer values and so we start counting at 0 so there is an implicit association of 0 with FALSE and 1 with TRUE which is a standard way to associate integers with truth values.

A struct on the other hand is a bundle of related data that can be passed around together. It is rather similar to an object in object-oriented programming languages such as Java. In this program there will be a single struct that you will be passing around, the player struct which looks like this:

```
struct player
{
    /**
     * the player's name
     **/
    char name[NAMELEN+1];
    /**
     * the color of the token for the player. Note: this is a
     * typedef of enum cell_contents.
     **/
    color thiscolor;
    /**
     * how many counters does this player have on the board?
     **/
    unsigned counters;
    /**
     * what type of player is this? A human or a computer?
     **/
    enum playertype type;
};
```

So, we can see from the comments that color is a typedef (alias) for enum cell_contents as this is kind of what the contents of each cell is – a colour. The player struct also stores the name, number of counters played by the player and their type – whether they are a human or computer player.

There are some other data structures used in the assignment as well but we will discuss these as we come to each assignment requirement.

**Further explanation of data structures will be available under each implementation requirement.**

## *Core Requirements (75%)*

The following are the core functional requirements of the assignment. That is, these requirements make up the basic functionality that you need to implement.

## 1. Main Menu (5 marks)

You are required to implement a main menu that manages the overall program. You should display the menu as follows:

```
Welcome to connect 4
--------------------
1. Play Game
2. Display High Scores
3. Quit
Please enter your choice:
```

Your menu input algorithm should reject any invalid data and should implement correct buffer handling. See the sections below on “buffer handling” and “input validation” for further information.

## 2. Data Structure Initialization (5 marks)

Various data structures used throughout your program need to be initialized to a known safe state. For example, you should initialize the two-dimensional game board to EMPTY, the name for each row of the scoreboard to the empty string, and so on. Leaving any data structure uninitialised means it is uncertain what the starting point for that element is and that is the source of many errors.

## 3. Player Setup (5 marks)

When a new game starts, you need to set up the structs to represent human and computer players.

For the human player, you need to ask for their name and store it in the “name” element of the player structure, whereas the Computer player's name should simply be set to “Computer”. You need to set the counters to an appropriate initial value. You will also need to randomly allocate a color to each player and set its type – whether it is a computer or a human player. Finally you need to set the player with white tokens as the current player as they always play first.

## 4. Play Game (10 marks)

This is the heart of the game.

Firstly, you need to initialise the game by calling functions written for requirement 3. The rest of this function is then managing the game loop.

Each player takes a turn to drop a token into a column on the game board. The task that needs to be performed will be different depending on whether this is a human or a computer player.

Regardless of the player type, you will need to display the board as it was before the player makes their move.

If the current player is a computer player, we just need to select a column at random and drop a token into that column. If there is no room, try another column.

If the current player is a human player, your program will need to ask the user what column they wish to drop a token in. This input should be validated to check that it is numeric and within the range of allowed columns.

Once a valid move has been entered, your program should update the gameboard. It should then check if the game has been won lost or drawn by calling and the change the current player.

The game loop is implemented in the function:

```
struct player * play_game( struct player * human,
     struct player * computer);
```

This function also checks if the game has been won, lost or drawn by calling `test_for_winner()`, and if the game is over, returns the winner to main or NULL if the game was a draw.

You should inform the user who won the game or if there was no winner before returning to the main menu.

## 5. Display Game Board (5 marks)

In this requirement you are required to display the game board. The game board should be displayed as shown on the first page of this assignment specification.

## 6. Player Turn (10 marks)

In this requirement, you need to handle the taking of a turn - either of a human or a computer player.

For the human player, you will need to allow the user to enter the column they wish to place a token in. You will need to validate what the user enters, then place a token correctly in that column so that it occupies the cell closest to the bottom of the board array for that column.

For the computer player, you will need to randomly generate a column number to place a token in and if that column is full, generate a different column number until a column number with some free space has been found.

For either player type, you need to increment the number of counters in the player struct after each successful turn. This is a count of how many tokens have been successfully played by that player in the current game.

The function prototype for the function to perform this task is:

```
enum input_result take_turn(struct player * current,
     enum cell_contents board[][BOARDWIDTH]);
```

Please see the “return to menu” requirement for more detail on the input result enumeration.

## 7. Swap Players (4 marks)

At the end of each turn, we need to change who the current player is as their turn has ended. We are going to do this by simply swapping the pointers for the current player and other player in the game. The function prototype for this function is:

```
void swap_players(struct player ** first, struct player **
second);
```

Where first and second are both pointers to pointers to a struct player.

For example, if current points to the human player and other points to the computer player, at the end of this function, current will point to the computer player whereas other will point to the human player.

This will be called from `play_game()` with a call such as `swap(&current, &other)`.

## 8. Test for Winner (10 marks)

In this requirement you are required to test what the game's current state is.

Possible game states are defined by the `game_state` enumeration:

```
enum game_state
{
    G_NO_WINNER=-1,
    G_DRAW,
    G_RED,
    G_WHITE
};
```

Most of these states should be self-explanatory but let's go through their meaning.

- `G_NO_WINNER`: the game is still in progress and therefore there is no winner yet.
- `G_DRAW`: neither player has won the game but there are no more spaces left for a player to make a move.
- `G_RED / G_WHITE`: the player whose token is the specified colour has won the game.

Your task in this function is to iterate over the 2-dimensional array (the board) looking for four in a row in any direction – if you find this, return the appropriate value of either `G_RED` or `G_WHITE`. Next, test for a draw. If none of these cases hold, return `G_NO_WINNER`.

## 9. Insert High Score (10 marks)

When the game ends, you need to return the appropriate game state back to main. You will then need to insert the winner's score sorted in order by the number of counters that they played in the game. You should only store the top ten scores and so when a new score is entered on a full scoreboard, the lowest score simply drops off the bottom of the board.

The function prototype for inserting a score is:

```
BOOLEAN add_to_scoreboard(scoreboard board,
     const score* lastscore);
```

Both scoreboard and score are typedefs (aliases) of other types.

Scoreboard is defined as:

```
typedef struct player scoreboard[SCOREBOARDSIZE];
```

and score is defined as:

```
typedef struct player score;
```

In other words, a `scoreboard` is an array of `struct player` of `SCOREBOARDSIZE` (10) and a score is another name of a player struct. This has been done so we can reuse the type and it simplifies the maintenance of the code.

## 10. Display High Scores (5 marks)

For this requirement, you will need to display the scores in the order they are scored in the scoreboard array. The display should look as follows:

```
Player              |Score
-------------------------------------------
Barney              |17
Magi                |15
Red                 |10
Computer            |8
Computer            |7
Computer            |6
Paul                |4
Fred                |4
Muaddib             |4
Zafiqa              |4
```

## 11. Return to Menu (4 marks)

This is a special requirement that has to do with proper buffer handling.

At any point in the running of this program, if the user presses the ctrl key and ‘d’ together or they press enter on a new line, they should be returned to the main menu.

Students often ask about how to deal with the ctrl-d combination and what it means. A little bit of research online (ie, following a couple of links from a search engine) should help you solve this problem. We want you to show some resourcefulness with this one.

In order to simplify this process, we have provided you with an enumeration that can be used with your user input functions to capture whether an input request has succeeded or failed. The definition of this enumeration is:

```
enum input_result
{
    SUCCESS,
    FAILURE,
    RTM=-1
};
```

So, when the input succeeds with a valid result, you can return `SUCCESS`, when it fails with an invalid result, you can return `FAILURE`, but if the user has fired a RTM event such as pressing enter of ctrl-d on a newline, you can return `RTM`.

## 12. Quit the Program (2 marks)

When the user selects quit from the main menu, your program should cleanly quit, without crashing.

## *13. General Requirements (20%)*

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 5, 5, 6 and 4 marks, respectively.

## *14: Demonstration (5%)*

This demonstration will occur in the week beginning 10 August 2015. You will need to demonstrate the following, without, at this stage, requiring to implement validation.

Ability to compile/execute your program from scratch (0.5 marks).

Requirements 1, 3, 5, and 6 implemented and running (1 mark each). That is, you will need to display the main menu, start a game, and implement the logic for computer and human players to be able to have a turn, including the display of the game board. Note that since we do not require implementation of the full game loop here. It is enough for each player to simply have one turn.

Submit your work (see Submission Instructions below) immediately after this demonstration (0.5 marks).

# Submission Information

## Submission date/time: Friday 28th August, 9 pm

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.

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:

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

**board.h**: contains the data definitions and function prototypes that relate to the manipulation of the game board that are not specifically related to the game rules.

**board.c**: contains the implementation of functions that relate to the manipulation of the game board that are not specifically related to the game board.

**bool.h**: contains the definition of the BOOLEAN type that may be used in various parts of your program.

**con4.h**: header file for data structures that need to be included in the main function.

**con4.c**: contains the main function that manages the running of the program.

**game.h**: contains the data structure definitions and function prototypes that relate to running the game.

**game.c**: contains the implementation of the functions involved in managing the game.

**helpers.h**: contains the data definitions and function prototypes for the helper functions such as those that manage input for your program.

**helpers.c**: contains the helper functions such as those that manage input for your program.

**player.h**: contains the data structure definitions and the function prototypes for manipulating player data.

**player.c**: contains the functions for manipulating player data

**scoreboard.h**: contains the data structure definitions and function prototypes for manipulating the scoreboard.

**scoreboard.c**: contains the functions for manipulating the scoreboard.

## Compilation Instructions:

In order to compile your program so that it meets the standards of this course you will need to compile it in this way with gcc:

```
gcc -ansi -Wall -pedantic board.c con4.c game.c helpers.c
    player.c scoreboard.c -o con4
```

## Sample Executable:

If you find some of these instructions confusing or you would like to see how they are supposed to run, we have provided you with an executable that you can compare your implementation against. You should make sure that your executable behaves as closely as possible to this reference executable. The reference executable is available on saturn, titan or jupiter from the following path:

```
/home/el9/E70949/shared/a1_sample/con4
```

## Submission instructions:

Create a zip archive using the following command on saturn or jupiter or titan:

```
zip $(USER)-a1 board.c board.h bool.h con4.c con4.h game.c \
game.h helpers.c helpers.h player.c player.h scoreboard.c \
scoreboard.h
```

This will create a .zip file with your username on the server, eg: if your student id is 3304456 then the file will be called: s3304456-a1.zip.

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

## Late Submissions:

10% of the possible marks for this assignment will be deducted for each day late and assignments submitted 5 or more days late will not be marked.

[END OF ASSIGNMENT #1 INFORMATION]