THE UNIVERSITY OF SYDNEY

COMP2129 Assignment 2

Atoms

Assignment Due: 8:00am Monday, 10 April, 2017

Morning Practical assessment to take place: 9:00am Monday, 10 April, 2017

Afternoon Practical assessment to take place: 4:00pm Monday, 10 April, 2017

This assessment is worth 8% of the course

Completion of this assignment alone counts towards 3% of your grade

Task Description

This assignment is based on a game called **Atoms**. You will be tasked with writing this game in the C programming language with dynamic data structures.

This game revolves around placing atoms on a gridded game board. Each grid space has a limit of how many atoms it can contain. Once the limit is reached, the atoms will expand to the adjacent spaces. This creates an interesting property in the game where chains can be triggered by a single placement.

The game will be accompanied by a number of utility functions for players to utilise such as saving, game statistics and loading. You will also need to implement a set of commands that will allow the players to interact with the game.

In summary:

- There is a game board that is **n** x **m** grid spaces.
- There is a minimum of **2** players and a maximum of **6**.
- Initially a grid space is unoccupied until a player places an atom in that grid space.
- A player can place an atom in a grid space owned by them or that is unoccupied.
- Once a grid space has hit its limit (number of atoms == limit), a single atom will expand out to the adjacent grid spaces.
- After the first k moves players can be removed from the game if they no longer own any grid spaces that player is then to be removed from the game.
- The user interacts with a set of commands that are specified later in this document
- The player colour order is **RGPBYW** (Red, Green, Purple, Blue, Yellow, White).

Working On Your Assignment

You can work on this assignment on your own computer or lab machines. It is important that you continually back up your assignment files onto your own machine, external drives and/or the cloud.

You are encouraged to submit your assignment on Ed while you are in the process of completing it. By submitting you will obtain some feedback of your progress on the sample test cases later provided.

During the practical assessment session you will only be able to submit your final solution within the time period allocated. You will be required to submit your assignment on Ed.

Academic Declaration

By submitting this assignment you declare the following:

I declare that I have read and understood the University of Sydney Academic Dishonesty and Plagiarism in Coursework Policy, and except where specifically acknowledged, the work contained in this assignment or project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the the Academic Dishonesty and Plagiarism in Coursework Policy, can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgement from other sources, including published works, the internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

I acknowledge that the School of Information Technologies, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and or communicate a copy of this assignment to a plagiarism checking service or in house computer program, and that a copy of the assignment may be maintained by the service or the School of IT for the purpose of future plagiarism checking.

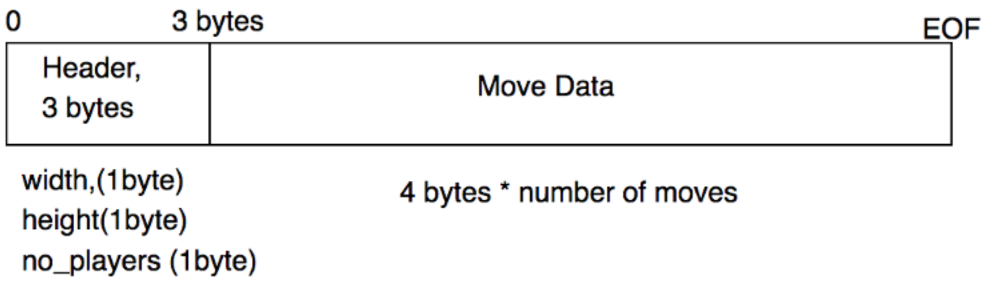
Implementation Details

Write a program in C that implements the game atoms. The sample test cases on ed will only contain valid input command, it is up to you to handle any invalid input. Commands are case sensitive.

Game rules and features:

- There are *k* players and each player takes a turn starting from player 1. there is a minimum of 2 players and a maximum of 6 players per a game
- A player can place an atom on a grid space that they already own or is unoccupied
  - If the grid space is a corner, then the limit is 2
  - If the grid space is on a side then the limit is 3
  - If the grid space is neither a corner or a side space then the limit is 4
  - (The pattern is the adjacent grid spaces)
- When a grid has reached its limit, the atoms will expand into the adjacent places. This can trigger other grids to reach their limit and expand.
- Players can undo moves that they have performed
- The maximum width of the board is 255, the minimum width is 2
- The maximum height of the board is 255, the minimum height is 2
- You may assume the maximum line length is 255
- There is an option to save the game and load it when the program has been reloaded

Save file specification is specified below:



The file header contains: 1 byte for width 1 byte for height 1 byte for player

After the header, the move data will fill the rest of the file until the end of the file.

Each move is encoded by 4bytes, First 2 bytes are used for coordinates where the empty two bytes encoded by 0 padding.

Below we have specified a header file with the structs that we suggest you use. You may modify these structs to suit your need.

The next page outlines the contents of atoms.h .

atoms.h

```
#ifndef ATOMS_H
#define ATOMS_H

#include <stdint.h>

#define MAX_LINE 255
#define MAX_PLAYERS 6
#define MIN_PLAYERS 2
#define MAX_WIDTH 255
#define MAX_HEIGHT 255
#define MIN_WIDTH 2
#define MIN_HEIGHT 2

typedef struct move_t move_t;
typedef struct player_t player_t;
typedef struct game_t game_t;
typedef struct grid_t grid_t;
typedef struct save_file_t save_file_t;

struct save_t {
  char* filename;
  save_file_t* data;
};

struct move_t {
  int x;
  int y;
  move_t* parent;
  move_t* extra;
  player_t* old_owner; //NULL if unoccupied
};

struct player_t {
  char* colour;
  int grids_owned;
};

struct game_t {
  move_t* moves;
};

struct grid_t {
  player_t* owner;
  int atom_count;
};

struct save_file_t {
    uint8_t width;
    uint8_t height;
    uint8_t no_players;
    uint32_t* raw_move_data;
};
#endif
```

Commands

HELP

Provides list of commands

```
HELP    displays this help message
QUIT    quits the current game

DISPLAY    draws the game board in terminal
START <width> <height> <number of players>    starts the game
PLACE <x> <y>    places an atom in a grid space
UNDO    undoes the last move made
STAT    displays game statistics

SAVE <filename>    saves the state of the game
LOAD    <filename>    loads a save file
PLAYFROM  <turn>  plays from n steps into the game
```

START <k> <width> <height>

The start command will accept 3 arguments, first argument is how many players are going to be in the game. The next two are the dimensions <width> and <height> of the gameboard.

This command needs to check the width * height >= n (number of players). Otherwise the program should respond with:

```
Cannot Start Game
```

If any of the input is invalid (width or height or n) are negative digits or character values, the program should respond with

```
Invalid command arguments
```

If the argument length is less than the required number (3) the program should specify

```
Missing Argument
```

The number of arguments is greater than the required number the program should respond with

```
Too Many Arguments
```

The START command cannot be used after the START command has been successfully executed. If this case is presented the program should respond with

```
Invalid Command
```

When START command is successfully executed the program should respond with, the colours are assumed to be in this order: RGPBYW (Red, Green, Purple, Blue, Yellow, White)

```
Game Ready
Red’s Turn
```

STAT

Displays the current state of the game

STAT cannot be executed unless a game has been started, if it is executed in this state the program should respond with

```
Game Not In Progress
```

When executed successfully the STAT command should display the current statistics related to the players.

```
Player Red:
Grid Count: 5

Player Green:
Grid Count: 2
```

In the account where a player has given up or has been removed from the game the STAT command should show. (In this example Player P gave up).

```
Player Red
Grid Count: 5

Player Green:
Grid Count: 3

Player Purple:
Lost
```

When printing out the the players, it will be from first index to last. The player colours are in the order: **RGPBYW** (Red, Green, Purple, Blue, Yellow, White).

PLACE <x> <y>

Places an atom at an x, y position on the game board that is associated with the player‘s turn

If the argument length does not equal two then the program should output Invalid Coordinates

If the x value is **less than** 0 or **greater than or equal to** the width and/or If the y value is **less than** 0 or **greater than or equal to** the height then the program should output

```
Invalid Coordinates
```

If the grid space selected is not owned by the player or is not unoccupied

```
Cannot Place Atom Here
```

In the case that PLACE command has not be executed successfully, the program should be ready for another command.

```
PLACE -1 -1
Invalid Coordinates

PLACE 0 0
Red's Turn
```

On success, the program should respond with whose move is next

```
PLACE 2 1
Red's Turn
```

In relation to expansion (when the grid space has reached its limit), the expansions move clockwise starting with the grid space that is y-1.

After a PLACE command is entered and only one player is remaining after it successfully executes, the program should respond with and quit:

```
PLACE 5 4
Red Wins
<program ends>
```

DISPLAY

Displays the gameboard using + to denote the corner and two – to denote the colour and the number of atoms located. Example Output

```
+--------------------+
|R1|  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |G1|G3|  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |R1|  |  |  |
+--------------------+
```

Empty spaces denote unknown grid spaces while <colour><number> (R1) denote the owner and the number of atoms in that grid space.

PLACE examples

```
PLACE 0 0
Green’s Turn

DISPLAY

+--------------------+
|R1|  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+--------------------+

PLACE 2 0
Red’s Turn

DISPLAY

+--------------------+
|R1|  |G1|  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+--------------------+

PLACE 0 0
Green’s Turn

DISPLAY

+--------------------+
|  |R1|G1|  |  |  |  |
|R1|  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+--------------------+

QUIT
Bye!
```

```
PLACE 0 0
Green’s Turn

DISPLAY

+--------------------+
|R1|  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+--------------------+

PLACE 2 0
Red’s Turn

DISPLAY

+--------------------+
|R1|  |G1|  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+--------------------+

PLACE 0 0
Green’s Turn

DISPLAY

+--------------------+
|  |R1|G1|  |  |  |  |
|R1|  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+--------------------+

PLACE 2 0
Red’s Turn

DISPLAY

+--------------------+
|  |R1|G2|  |  |  |  |
|R1|  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+--------------------+
PLACE 1 0
Green's Turn

DISPLAY

+---------------------+
|   |R2|G2|  |  |  |  |
|R1|  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+---------------------+

PLACE 2 0
Red's Turn

DISPLAY

+---------------------+
|G1|  |G1|G1|  |  |  |
|R1|G1|G1|  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+---------------------+

PLACE 0 1
Red's Turn

DISPLAY

+---------------------+
|G1|  |G1|G1|  |  |  |
|R2|G1|G1|  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+---------------------+

PLACE 0 0
Green Wins

QUIT
Bye!
```

UNDO

Once a player has made a move, it is up to the will of the next player to let the other player undo it. We are to pretend that this game involves hotseating and therefore it is agreed upon the players if they are to allow an undo.

```
DISPLAY

+--------------------+
|  |  |  |  |  |  |  |
|  |  |  |P1|  |  |  |
|  |G1|  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |R1|  |  |  |
+--------------------+

PLACE 2 1
Red's Turn

+--------------------+
|  |  |  |  |  |  |  |
|  |  |B1|P1|  |  |  |
|  |G1|  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |R1|  |  |  |
+--------------------+

UNDO
Blue's Turn

+--------------------+
|  |  |  |  |  |  |  |
|  |  |  |P1|  |  |  |
|  |G1|  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |R1|  |  |  |
+--------------------+
```

In the case that we perform UNDO at start of the game, the program should output

```
Cannot Undo
```

QUIT

Quits the game early

```
QUIT
Bye!
```

This quits the game early.

SAVE <filename>

Save the current state of the game with a filename

You will need to adhere to a strict file structure, the file is saved as a binary file. Where game information is stored in the header of the file and the leading data is the move set.

Use the following struct

```
struct save_file_t {
    uint8_t width;
    uint8_t height;
    uint8_t no_players;
    uint32_t* raw_move_data;
};
```

width, height and no_players are encoded as unsigned bytes, by reading 3 bytes from the file, you will be able to access the width, height and no_players.

After the header has been read, the rest of the file can be assumed to be raw_move_data, each move is encoded as a 32bit integer (4 bytes). Each 32bit integer can be extracted as x (1 byte), y (1 byte) coordinates with 2 bytes of 0 padding afterwards.

When creating a save file, we do not need to include any move that has been undone.

This command creates a file and saves the current game details and move set, if a file already exists with this name the function should reject the save file.

```
File Already Exists
```

On success the program should output

```
Game Saved
```

LOAD <filename>

Load command takes a filename as an argument, if that file does not exist the program should remain in its original state and output.

```
Cannot Load Save
```

If a game has already started or load has been executed successfully, subsequently load commands should not attempt to load a new game and instead the program should respond with:

```
Restart Application To Load Save
```

If the command is executed successfully the program should respond with:

```
Game Loaded
```

If any other command besides QUIT or PLAYFROM after the LOAD command has been executed successfully, the program should output:

```
Invalid Command
```

PLAYFROM <turn>

This command allows the user to play from a turn **n** when a file has been loaded.

If the turn specified is > the number of turns loaded from the file the program should consider this as the last turn or if the turn argument is instead "END"

```
PLAYFROM END
Game Ready
Yellow's Turn
```

If the turn specified is < 0

```
Invalid Turn Number
```

If PLAYFROM is executed prior to a LOAD command the program should ignore it and respond:

```
Invalid Command
```

If the turn specified is > 0 and <= the number of moves made, the program should output that the game is ready and what player's turn it is.

```
Game Ready
Yellow's Turn
```

Writing Test Cases

Samples testcases will be made available but these do not test the entirety of the program specification. You will need to write your own test cases to ensure that your program covers as many cases as possible.

Test cases should be placed in the tests/ directory provided with the code scaffold. You need to provide input and output files. Typically this is in the form of **testcase.in** (for input) and **testcase.out** (for output).

Try to name your test cases to be brief but descriptive of the case they are testing.

Submission Details

You must submit your code prior to the Week 6 Practical Assessment. You are required to attend the practical assessment on Monday Week 6 (9:00am Monday, 10 April) to complete the assignment.

Your code and tests must be submitted using Ed. You can upload your files in the assessment page of the appropriate assessment.

You are encouraged to submit multiple time, but only your last submission will be marked.

Marking

Your program will be marked automatically by Ed, Please ensure that you carefully follow the assignment specification. Your program must match the exact output in the examples and the test cases on Ed.

3 marks are assigned based on automatic tests for correctness of your program

5 marks are assigned based on automatic tests for correctness of your program in the practical assessment session

Warning: Any attempts to deceive or disrupt the marking system will result in an immediate zero for the entire assignment. Negative marks can be assigned if you do not properly follow the assignment specification, or your code is unnecessarily or deliberately obfuscated.