The University of Queensland
School of Information Technology and Electrical Engineering
Semester One, 2020
CSSE2310 / CSSE7231 - Assignment 3
Due: 20:00pm 8th May, 2020
Marks: 50
Weighting: 25% of your overall assignment mark (CSSE2310)
Revision 3.0

Integrity

This is an individual assignment. You should feel free to discuss aspects of C programming and the assignment specification with fellow students. You should not actively help (or seek help from) other students with the actual coding of your assignment solution. It is cheating to look at another student's code and it is cheating to allow your code to be seen or shared in printed or electronic form. You should note that all submitted code may be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct proceedings will be initiated against you. A likely penalty for a first offence would be a mark of 0 for the assignment. Don't risk it! If you're having trouble, seek help from a member of the teaching staff. Don't be tempted to copy another student's code.

Do not commit any code to your repository unless it is your own work or it was given to you by teaching staff. **If you have questions about this, please ask.**

While you are permitted to use sample code supplied by teaching staff this year in this course. Code supplied for other courses or other offerings of this course is off limits — it may be deemed to be without academic merit and removed from your program before testing.

Purpose

In this assignment, you will write a group of three C99 programs to play a game (described later). One of the programs will be called `2310dealer` it will control the game. The other two programs (`2310A` and `2310B` will be players.

The dealer will be responsible for running the player processes and communicating with them via pipes (created via `pipe()`). These pipes will be connected to the players' standard ins and outs so from the players' point of view, communication will be via `stdin` and `stdout`.

Other than the dealer starting player processes, there should not be any other parallelism used in this assignment. For example, no multi-threading nor non-blocking operations. Your programs are not permitted to start any additional processes[1] nor are your programs permitted to create any files *during their execution.*

Your assignment submission must comply with the C style guide (version 2.0.4) available on the course blackboard area. It must also not use banned functions or commands listed in the same place.

Game

What follows is an abstract description of the game. Precisely what your programs need to output will be described separately.

The game is played using a sequence of sites called the path. Each site has a type and a limit of the number of players that can visit that site. **Each player starts the game with seven money**. Players take turns moving forward some number of steps, the type of site they end on determines what action they take.

[1]other than those specified on the dealer commandline arguments

Site types are:

Site	**Action taken by player**
Mo	gain 3 money.
V1	gain 1 point for each V1 site they have visited before this one.
V2	gain 1 point for each V2 site they have visited before this one.
Do	Convert all money into points (1 point for each 2 money spent; rounded down).
Ri	draws the next card from the item deck.
::	No action. But all players must stop at this site. (This type of site will be referred to as a barrier)

Each card in the item deck is represented by a single character {A, B, C, D, E}.

Turn order

At the beginning of the game, players will be arranged with the smallest player-id furthest from the path; in increasing order so the highest player-id is closest to the path. eg:

```
:: Mo1 Do1 Ri1 Ri1 ::
   1
   0
```

Note that:

- each site label is followed by a space (including the final barrier).
- all lines in the path display should have trailing spaces added as necessary to make them the same length as the top line.

The next player to make their move is the one furthest back. This means that a player which makes smaller steps may get more turns. The largest move a player can make is to move directly to the next barrier.

For example:

```
:: Mo1 Do1 Ri1 Ri1 ::
   0       1
```

Player 0 is furthest back so it is their turn. Player 0 moves one step.

```
:: Mo1 Do1 Ri1 Ri1 ::
       0   1
```

Player 0 is furthest back so it is their turn. Player 0 moves two steps.

```
:: Mo1 Do1 Ri1 Ri1 ::
           1   0
```

Player 1 is furthest back. Player 1 moves two steps.

```
:: Mo1 Do1 Ri1 Ri1 ::
               0   1
```

Player 1 moves one step.

```
:: Mo1 Do1 Ri1 Ri1 ::
                   1
                   0
```

Sites with limit >1

Some sites will allow multiple players to visit. For turn order purposes, players at a site take turns in order or most recent arrival. Barrier sites are assumed to have enough capacity to hold all players.

```
:: Do1 Ri1 Ri1 :: Mo1 Mo1 ::
       0   1   2
```

Player 0's turn.

```
:: Do1 Ri1 Ri1 :: Mo1 Mo1 ::
           1   2
               0
```

Player 1's turn.

```
:: Do1 Ri1 Ri1 :: Mo1 Mo1 ::
               2
               0
               1
```

Player 1's turn.

```
:: Do1 Ri1 Ri1 :: Mo1 Mo1 ::
               2  1
               0
```

Player 0's turn.

```
:: Do1 Ri1 Ri1 :: Mo1 Mo1 ::
               2  1   0
```

Player 2's turn.

```
:: Do1 Ri1 Ri1 :: Mo1 Mo1 ::
                  1   0   2
```

Player 1's turn

```
:: Do1 Ri1 Ri1 :: Mo1 Mo1 ::
                      0   2
                          1
```

Points from items

At the end of the game, players gain points based on the item cards them have collected.

- Each complete set of ABCDE is worth ten points.
- Each set of four types is worth seven points.
- Each set of three types is worth five points.
- Each set of two types is worth three points.
- A single type is worth one point.

For example:

- AAABCCDE → ABCDE + AC + A = 10 + 3 + 1
- AABBBCCDDDEE → ABCDE + ABCDE + BD = 10 + 10 + 3

2310dealer

The dealer will take the following commandline arguments (in order):

- the name of the file to read the item deck from.
- the name of the file to read the path from.
- one or more player programs to start

Eg: `./2310dealer d1.deck 1.map ./X ./X ./Y`
Would start a game with two `X` players and a `Y` player.

Note: your dealer should run whatever program names it is given. Do not try to adjust your dealer's behaviour based on the names it is passed.

When running player processes, the dealer must ensure that any output to `stderr` by players is supressed.

As soon as the dealer reads a ˆ from a player process, the dealer should send the path to that player.

When the dealer receives `SIGHUP`, it should kill and reap any remaining child processes. Note: We won't test the exit status for `2310dealer` when it receives `SIGHUP`.

Deck file format

Deck files consist of a single line beginning with the number of cards in the deck. The cards immediately follow the number and are represented by the characters 'A'. . . 'E' (inclusive). The line must have at least four cards in it and must be terminated by '\n'.

For example:

```
7ABACDEE
```

Note: We won't test for leading or trailing whitespace in this file.

If the dealer runs out of cards in the deck, go back to the beginning of the deck and keep going.

Path file format

The file consists of a single line of text ('\n' terminated). The line begins with the number of stops in the path followed by a semi-colon. The rest of the line represents the path.

Barriers will be represted as `::-`. All other types are encoded as type {Mo, V1, V2, Do, Ri, Si} followed by capacity (a number between 1 and 9 inclusive) The path must start and end with `::` sites.

For example: `7;::-Mo1V11V22Mo1Mo1::-`

Dealer output

All of the following output is to `stdout`.

At the beginning of the game the dealer will display the path. After each *valid* move recieved from a player, the dealer will print information about the moving player:

- `Player ? Money=? V1=? V2=? Points=? A=? B=? C=? D=? E=?`
  substitute in the current amount of each the player has.

Then redisplay the path (see Turn order for how to display the path).

At the (normal) end of the game, print out the scores in player order (comma separated, newline terminated). eg:

```
Scores: 5,3,5
```

Dealer errors

All error output to be sent to `stderr`. Check the following conditions in order:

Exit	Condition	Message
0	Normal end	
1	Incorrect number of args	`Usage: 2310dealer path p1 {p2}`
2	Invalid path file or contents	`Error reading path`
3	Invalid deck file or contents	`Error reading deck`
4	Error starting one of the players	`Error starting process`
5	Communications error	`Communications error`

The hub should check whether the site a player wishes to move to is valid, including if it would move past a barrier site.

Players

All player processes take the following commandline arguments (in order):

- Number of players
- This player's ID (starting at 0)

Once the player has checked its commandline arguments, it should:

1. Print a caret(^) to its standard out (no newline following it).
2. Read the path from `stdin` (in the same format as the path file).

Regarding strategy: you must always check to see if there is a "barrier" site between the player and the site they would have chosen. The player must not move past a barrier in order to reach their destination. For the purposes of strategy, sites past the next barrier should be ignored. When looking for closest sites of a given type, ignore any sites which are full.

Player A strategy

In the following, Check the following in order, stop when a condition matches.

- The player has money, and there is a `Do` site in front of them, go there.
- If the next site is `Mo`, then go there.
- Pick the closest `V1`, `V2` or `::`, then go there.

Note that this player does not visit `Ri` sites. Your players should still be able to cope with other players visiting them.

Player B strategy

In the following, where there is more than one possible match, choose the closest one.

- If the next site is not empty and all other players are in front of us, move forward one site.
- If we have an odd amount of money, and there is a `Mo` between us and the next barrier, then go there.

- If we have the most cards or if everyone has zero cards and there is a `Ri` between us and the next barrier, then go there.
- If there is a `V2` between us and the next barrier, then go there.
- Move forward one site.

Player output

Players should produce the same output as described in the "Dealer output" section, but must send it to `stderr` instead of `stdout`.

Player errors

All error output should be sent to `stderr`. Check the following conditions in order:

Exit	**Condition**	**Message**
0	Normal end	
1	Incorrect number of args	`Usage: player pcount ID`
2	Player count is < 1	`Invalid player count`
3	Player-id is < 0 or $\geq$ playercount	`Invalid ID`
4	Error in path	`Invalid path`
5	Early game over	`Early game over`
6	Communications error	`Communications error`

Communications errors from the player's point of view include:

- unexpected EOF on `stdin`
- messages not being in the correct format
- messages that describe non-existant players or site numbers.

We will not test other types of sematically invalid content (eg moving backwards or gaining points on the wrong type of site) to players. You are free to flag such messages as invalid if that makes your coding easier.

Messages

Direction	**Format**	**Detail**
dealer → player	`YT`	Player should send back a move
player → dealer	`DO`*n*	Player want to move to site *n* (begining at zero)
dealer → all	`EARLY`	Game ended early due to comms error
dealer → all	`DONE`	Game ended normally
dealer → all	`HAP`*p*`,`*n*`,`*s*`,`*m*`,`*c*	A player made a move. *p*=player-id *n*=new site for player *s*=additional points for player *m*=additonal money for player *c*=card drawn by player

Compilation

Your code must compile (on a clean checkout) with the command:
`make`

Each individual `.c` file must compile with at least `-Wall -pedantic -std=gnu99`. You may of course use additional flags but you must not use them to try to disable or hide warnings. You must also not use pragmas to achieve the same goal. Your code must be compiled with the `gcc` compiler.

If the make command does not produce one or more of the required programs, then those programs will not be marked. If none of the required programs are produced, then you will receive 0 marks for functionality. Any code without academic merit will be removed from your program before compilation is attempted [This will be done even if it prevents the code from compiling]. If your code produces warnings (as opposed to errors), then you will lose style marks (see later).

Your solution must not invoke other programs apart from those listed in the command line arguments for the hub. Your solution must not use non-standard headers/libraries.

Submission

No late submissions will be marked for this assignment under any circumstances. Submission must be made electronically by committing using subversion. In order to mark your assignment, the markers will check out `/trunk/ass3/` from your repository on `source.eait.uq.edu.au`. *Do not create subdirectories under /trunk/ass3/.* The marking may delete any such directories before attempting to compile. Code checked in to any other part of your repository will not be marked.

Test scripts will be provided to test the code on the trunk. Students are *strongly advised* to make use of this facility after committing.

Note: Any .h or .c files in your `trunk/ass3` directory will be marked for style *even if they are neither linked by the makefile nor #included by some other file.* If you need help moving/removing files in svn, then ask. Consult the style guide for other restrictions.

You must submit a ***`Makefile`*** *or we will not be able to compile your assignment.* Remember that your assignment will be marked electronically and strict adherance to the specification is critical.

Marks

Marks will be awarded for both functionality and style.

Functionality (42 marks)

Provided that your code compiles (see above), you will earn functionality marks based on the number of features your programs correctly implement, as outlined below. If only some of your programs compile, the ones which compile will be marked. Partial marks may be awarded for partially meeting the functionality requirements. Not all features are of equal difficulty. If your program does not allow a feature to be tested then you will receive 0 marks for that feature, even if you claim to have implemented it. For example, if your program can never open a file, we can not determine if your program would have loaded input from it. The markers will make no alterations to your code (other than to remove code without academic merit). Your programs should not crash or lock up/loop indefinitely. Your programs should not run for unreasonably long times.

- 2310A player
  - Arg checking (2 marks)
  - Initial path output (2 marks)

  - Making first move (2 marks)
  - Complete games as solo player (2 marks)
  - Complete games with multiple players (2 marks)
  - Early disconnect and coms error handling (3 marks)
- 2310B player
  - Arg checking and initial path output (2 marks)
  - Initial move (2 marks)
  - Complete games (4 marks)
  - Early disconnect and coms error handling (1 marks)
- For dealer
  - Argument checking (2 marks)
  - Starting player checks (2 marks)
  - Complete game with only 2310A players (5 marks)
  - Complete game with unspecified players (7 marks)
  - Comms error handling, early disconnect and `SIGHUP` handling (2 marks)

Style (8 marks)

Style marks will be calculated as follows:
Let A be the number of style violations detected by simpatico plus the number of build warnings. Let H be the number of style violations detected by human markers. Let F be the functionality mark for your assignment.

- If $A > 10$, then your style mark will be zero and M will not be calculated.
- Otherwise, let $M_A = 4 \times 0.8^A$ and $M_H = M_A - 0.5 \times H$ your style mark S will be $M_A + \max\{0, M_H\}$.

Your total mark for the assignment will be $F + \min\{F, S\}$.

Specification Updates

It is possible that this specification contains errors or inconsistencies or missing information. It is possible that clarifications will be issued via the course website. Any such clarifications posted 5 days (120 hours) or more before the due date will form part of the assignment specification. If you find any inconsistencies or omissions, please notify the teaching staff.

Tips and Fixes

1. Your work will be made much easier if you place code used in multiple places into a source file shared by multiple programs. eg:
   - Encoding and decoding messages
   - Where is the next site of type X?
2. Write some test code to check shared functions *before* using them in the programs.
3. Remember that there is no '\n' after the initial '^'
4. Do not make your players' behaviour depend on argv[0]. There is no guarantee that programs will be named the way you think they will.
5. Use `tee` to capture the input and output to your players for debugging purposes.
6. Messages not terminated by '\n' (ie EOF happens), should still be processed.
7. You are not permitted to use non-blocking I/O in this assignement.
8. You are not permitted to use multi-threading in this assignment.
9. We strongly recommend not using `read()` or `write()` in this assignment `fread()` and `fwrite()` are better but still not as useful as you might think.
10. Start work on the players (remember you can feed input to them via standard in).