Assignment 1 - LRC Report

Jackson C. Dawson

CSE 13S - Winter 24

Purpose

The program "lrc.c" simulates gameplay of the popular card game, "Left Right Center". The user is initially prompted to decide how many "players" will participate in the game, and to provide a seed to use for simulating random dice rolls. Then, the program initializes the inputted number of players with three chips each, and has each "player" engage in a faux game of Left-Right-Center. Unlike in the real card game, gameplay starts with player 0, rather than the youngest player. During any given player's turn, they must roll as many dice as chips they have, with a limit of three dice. There are four possible outcomes of their roll- DOT, LEFT, RIGHT, or CENTER- with probabilities of 3/6, ⅙, ⅙, ⅙ respectively. Upon rolling a dot, the player does nothing. A left dictates that the player must pass one chip to their left, and a right to their right. Rolling center requires the player to put one chip in the "pot". After each player's turn the program will print the number of chips that the player ended their turn with. Gameplay continues until only one player has any chips left, and finally a winner is printed and the program returns 0.

Randomness

Randomness, as I see it, is the probability of something happening such that it cannot be accurately predicted with any level of certainty. If one is able to determine a method to accurately predict the outcome of the event, it is not truly random. I think that true randomness exists in terms of the universe when considering examples such as entropy? But in the domain of a computer, I think that true randomness is extremely difficult to simulate, if not impossible. We are using pseudo random numbers for this assignment because computers require steps to take in order to function, and such steps can be worked backwards methodically. Thus they are not truly random.

What is an abstraction?

An abstraction is, as I understand it, essentially "hidden complexity". It is the hiding of something complicated and only showing what is absolutely necessary to provide a cleaner or more usable end result. For example, many of the mechanics behind a pinball machine are not visible to the player. These components have been "abstracted" in order to create a better and easier experience for the end user.

Why?

Just as the middle schooler appreciates (perhaps unwittingly) the abstraction of the guts inside their favorite arcade game, programmers enjoy appropriate levels of abstraction as it makes our code easier to understand and maintain! Abstractions allow us to break what could be long stretches of code into bite-size chunks that can be understood far more easily. So it makes it easier to understand which component does what. Additionally, abstraction can help to point to the exact point a program is failing due to more defined behavior. Another use I can think of is libraries. It seems like we'll be using a lot of stdio.h, which (I think) is a library of functions that facilitate input and output in programs. We we #include <stdio.h>, and perhaps use printf(), we are using a function which has been abstracted away from us. There is code executing behind printf(), even if we may not be able (or want) to look at it.

Functions

For this assignment, two functions that might be particularly useful could be: 1. isWinner: A function which checks the chip count for all players and determines whether or not there is a winner ; 2. rollDie: A function which generates a random number between 0 and 5 in order to represent a single dice roll. Some additional functions that may or may not be beneficial in practice are: 3. playRound: A function which executes the gameplay for one single round ; 4. giveChipLeft: A function which executes if the die says "LEFT", which adds one chip to the player's opponent on their left ; 5. giveChipRight: A function which does the same thing as the previous except for the right opponent ; 6. decrementChips: To be used in tandem with the previous two, decrements the amount of chips the current player has by 1 ; 7. incrementChips: To call on neighboring players when they are due to receive a chip ; 8. initPlayers: To be called before gameplay starts- initializing an array with the appropriate indices and creating a player object for as many players as are needed.

Testing

I want to test a variety of inputs against the results of the same inputs given to the provided binary implementation. I want to test a few different valid numbers for seeds, and a couple different valid entries for the number of players. Then I want to check some edge cases; a few different non-valid inputs for both seeds ("a", "&") and player_num ("a", "&"). All tests will of course be diffed against the binary implementation provided in resources.

Putting it all together

Using a pseudorandom number generator allows us to expect consistent outcomes with consistent seeds. This will be extremely important in order for our work to be graded, and also for us to test our work against a correct implementation. Abstractions will help to both simplify the design process and make debugging an easier process.

How to Use the Program

After logging into the VM via SSH, navigate to the asgn1 directory:

```
~$ cd ./jcdawson/asgn1
```

Then use the following commands to format and compile the program:

```
~$ make format
```

```
~$ make
```

Now the program is ready to run! Call ~$ ./ lrc to run the compiled program.

During runtime, you will be prompted to input a number between 3 and 10 to represent the number of players, and then to enter an integer-seed for random die-roll generation. If you do not enter a valid number of players or seed, the program will continue executing with the default settings.

Finally, make sure to run the following command to clean up the directory:

```
~& make clean
```

That's it. You're all done!

Program Design

Currently, the Position enum and die array are the first pieces to be initialized. Then the struct defining a Player is created. Each player object has a "char name" and "int chips" attribute. After these lines, the Main function executes. Input is taken in from the player and validated; default settings are used if the respective input is found to be invalid. Then, the "players" array is initialized and filled according to the amount dictated by the user. The main game behavior is executed using a while loop, which checks the value of a function called isWinner, and confirms that there is not a winner using the "!" operator. This function uses a for loop to count the number of players who have at least one chip, and returns true if only one player has chips.

Within the while loop, a for loop is used to execute each player's turn, starting from player 0. A nested for loop executes code to roll the die for each chip that the player has with a maximum of 3. This is currently not abstracted into a separate function, but likely will be for my final implementation. At the end of the outer for loop, the player's ending chip count is printed. As soon as the while loop breaks, a function (yet to be written) will loop through to find the winning player and print their victory.

Pseudocode

```
// Define Position Enumerations
enum Position: DOT, LEFT, RIGHT, CENTER

// Initialize die array with Position values
die = [DOT, DOT, DOT, LEFT, RIGHT, CENTER]

// Define Player structure
structure Player:
  name
  chips

// Main function
function main():

  // Initialize number of players and prompt user for input
  num_players = 3
  display "Number of players (3 to 10)? "
  scanf_result = read input for num_players

  // Validate number of players input
  if scanf_result is invalid or num_players < 3 or num_players > 10:
    display error "Invalid number of players. Using 3 instead."
    num_players = 3

  // Initialize and prompt for random seed
  seed = 4823
  display "Random-number seed? "
  scanf_result = read input for seed

  // Validate seed input
  if scanf_result is invalid:
    display error "Invalid seed. Using 4823 instead."
    set random seed to SEED
  else:
    set random seed to seed

  // Create and initialize players array
```

```
    players = array of Player with size num_players
    for i from 0 to num_players:
        player = new Player
        player.name = player_name[i]
        player.chips = 3
        players[i] = player

    // Main game loop
    while not isWinner(players, num_players):
        for each player in players:
            // Execute turn for each player
            for each chip in player (max 3):
                roll = random() % 6
                if roll is DOT:
                    do nothing
                if roll is LEFT:
                    give a chip to the player on the left
                if roll is RIGHT:
                    give a chip to the player on the right
            display "player: ends their turn with {player.chips} chips"

    // Determine and display winner
    winner = findWinner(players, num_players)
    display "{winner's name} won!"

    return 0

// Function to check if there is a winner
function isWinner(players, num_players):
    players_with_chips = 0
    for each player in players:
        if player.chips > 0:
            players_with_chips++
    return players_with_chips == 1

// Function to find the winning player
function findWinner(players, num_players):
    for each player in players:
        if player.chips > 0:
            return player
```

Function Descriptions

Function: main

Inputs: The function implicitly takes input from the user when prompted, specifically the number of players and the random number seed.
Outputs: It outputs to the console the prompts for the user and the game's results, but returns an integer signifying the exit status of the program (commonly 0 for successful completion).
Purpose: Serves as the entry point for the program, where it initializes the game, manages the game loop, and terminates upon identifying a winner.

Function: isWinner

Inputs: An array of Player structures and an integer num_players representing the number of players.
Outputs: It outputs an integer value that is 1 (true) if there is a winner and 0 (false) otherwise.
Purpose: Determines if there is a single winner by checking if only one player has chips left.

Function: findWinner

Inputs: An array of Player structures and an integer num_players representing the number of players.
Outputs: Returns the Player structure that represents the winning player.
Purpose: Identifies and returns the winning player who has chips left when all others have none.