Assignment 2
LCR — Left/Center/Right Game

by Jessie Srinivas
Based on assignments by Dr. Darrell Long
edits by Dr. Kerry Veenstra
CSE 13S, Spring 2024
Document version 2 (changes in Section 10)

Due Thursday April 25th, 2024, at 11:59 pm
Draft Due Tuesday, April 23rd, 2024, at 11:59 pm

1 Introduction

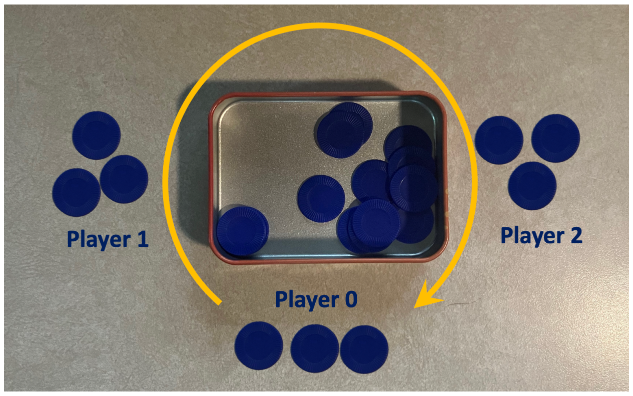
We are going to simulate[1] a simplified version of the dice game Left, Center, Right[2]. This game is entirely a game of chance, with no skill or player decisions (except for a die roll). It has relatively simple rules, which makes it easy to implement in C. Fig. 1 shows a game of Left-Center-Right with three players.

2 The Rules of the Game

In this simulation, the user will specify the number players in the game. The names of the players will be given to you. The first player in the list of names will go first. Players will sit in a circle in the order that they appear in the list of names with a pot in the middle of the table. Each player will start with 3 chips, and roll one die for each chip, with a maximum of three dice rolled per turn. A player will then roll their 6 sided dice, and do something based on the result of those rolls. For each die, they will follow the instructions in the list below.

[1]We are *simulating* it, and not creating it because we don't actually let players play the game, you simply watch it happen.
[2]One version of the rules is described in this YouTube video `https://www.youtube.com/watch?v=OxxpTWk3Rx0`

Figure 1: Left, Center, Right Game



To avoid confusion when indexing into an array, we are going to use a "CSE 13S die" which has numbers 0 through 5.

- If the player rolls a $\boxed{0}$, $\boxed{1}$, or $\boxed{2}$, (this is called a `DOT`) the player will keep one chip.
- If the player rolls a $\boxed{3}$, (this is called `LEFT`) the player will pass one chip to the left (next higher player number, or `0` if current player is `num_players - 1`).
- If the player rolls a $\boxed{4}$, (this is called `CENTER`) the player will place one chip in the pot.
- If the player rolls a $\boxed{5}$, (this is called `RIGHT`) the player will pass one chip to the right (next lower player number, or `num_players - 1` if current player is `0`).

After that turn, the next player will take their turn. Some more details:

- The game stops when only one player has chips. That player is the winner.
- If a player has no chips, they are still in the game, but do not get to roll a die. They can begin rolling again if they are passed a chip by one of their neighbors.

3 Game Abstractions

One of the most important skills to polish as Computer Scientists is the ability to create abstractions for the problems that we need to solve. In this program, you will be using an enumeration for the die.

3.1 Enumerating the Positions

In this game, we differentate between the "roll" of a die and its "position."

That is, when you roll the die, it can have any of the values $\boxed{0}$, $\boxed{1}$, $\boxed{2}$, $\boxed{3}$, $\boxed{4}$, or $\boxed{5}$. However, the game talks about die "positions," which are `DOT`, `LEFT`, `CENTER`, and `RIGHT`. While it is possible to represent these positions through their *roll* values from 0 through 5 and then check what the roll is with 6 if statements, the *position* of the die can be better represented with an abstraction.

First, rolling a $\boxed{0}$, $\boxed{1}$, or $\boxed{2}$ gives the same position: a `DOT`. The positions of the other three rolls also have names, which may be confusing to keep track of while coding. To make this simpler, we can first create an `enum` (enumerated) type. This allows us to create names for the die positions that we can use in the code. The syntax for this is as follows:

```
typedef enum {DOT, LEFT, CENTER, RIGHT} Position;
```

Let's break that down ...

First, `typedef`. This means that we are defining a type. The syntax for this is `typedef` *something name*`;`. Skipping to the end, that name is `Position`. This means that we can now use `Position` in our code to represent the *something* in the middle. Now, what's the *something*? This is the type that we will call `Position`. While it is possible to use an already existing type, we want to create a new one. The type we create is `enum { DOT, LEFT, CENTER, RIGHT }`, meaning that it is an enumerated type that contains 4 names, which are the names of the dice positions. While internally, these names are just equivalent to numbers, it is much easier to use the names in your code. Putting it all together, we can now create a position variable, and use it in standard C syntax. Two examples of how this could be used are as follows:

```
Position roll = DOT;

if (roll != LEFT) {
    /* do something */
}
```

The next thing we need to do is create an array that represents a die. This array will let us convert from a *roll* (with a value between 0 and 5) into a *position* (with values `DOT`, `LEFT`, `CENTER`, and `RIGHT`.) The syntax for that (and the sides) are as follows:

```
const Position die[6] = {
    DOT,
    DOT,
    DOT,
    LEFT,
    CENTER,
    RIGHT,
};
```

Your die **MUST** be the same as this one for your code to work. A few notes on this syntax.

1. This array is defined with the `const` keyword. This means that it cannot be modified after it is created. This is useful to make sure that you don't mess up the die.

2. Each of this array's elements has type the type `Position`. We can do this because of the enum definition above.

3.2 Generating Pseudorandom Numbers

To generate the pseudorandom numbers that are required to simulate the rolling of the dice, you will need a pseudorandom number generator (PRNG). The PRNG that you use will generate values between 0 and 999999. For example, the first three numbers of its default sequence are 211327, 653794, and 122201.

After you create a pseudorandom number, to make it work well with the die array defined above, you should use some function to force the random number to be in the range $0 \leq$ `roll` ≤ 5. (Hint: look at the modulus operator in K&R section *2.5 Arithmetic Operators* and section *A7.6 Multiplicitive Operators*.) If you follow this hint, then the first three die rolls that are implied by the sequence above will be 1, 4, and 5.

You will use the PRNG that we provide you. (The standard C library also provides you with a PRNG, but we don't use it because for grading purposes we want to be certain that your program uses the same pseudo-random sequence of numbers on your laptop as we will get on the grading computer, and sometimes a student will try to use a different version of an operating system, which has a different PRNG.)

Our PRNG is interfaced with the two functions `cse13s_random_seed()` and `cse13s_random()`. In order to use these functions, you will need to "include" a file in your program using this line:

```
#include "cse13s_random.h"
```

The `cse13s_random_seed()` function is essential in order to make your program reproducible. The function `cse13s_random_seed()` sets the random seed, which effectively establishes the start point of the pseudorandom number sequence that is generated. This means, after calling `cse13s_random_seed()` with a seed, that the pseudorandom numbers that are generated by `cse13s_random()` always appear in the same order.

4 Your tasks

You will be performing three main tasks:

1. Design your program and test cases (documented in `design.pdf`).

2. Write your program in the C programming language (in `lcr.c`).

3. Update your design to reflect the final version of your code.

I'm distinguishing between the first two tasks because they really are separate. The first task is to understand the data and the algorithms that are needed to, in this case, simulate the LCR game as described in the rules of Section 2. You confirm your understanding of the data and algorithms of that design by writing a design report describing what you know. Then, when you have an idea of what you want to do, you translate that idea into the C programming language.

As encouragement to start on your design, you will be submitting a draft of your design early! For points! (See the Assignment on Canvas.) Turning in your draft means following the sequence add / commit / push / submit commit ID for your draft document before the draft's deadline.

Yes, this assignment has two deadlines.

After you have a draft of your design, you write your program, placing the implementation in the source code file lcr.c. You will find starter files for this assignment in the course resources repository on git.ucsc.edu. The starter files are names.h and Makefile.

Finally, you submit a final design report that mentions any unintended shortcomings, and how you tested your code. You submit this final design report along with your source code by the final assignment deadline.

5 Program Structure

The structure of your program should follow these steps:[3]

1. Prompt the user to input the number of players, scanning in their input from `stdin`. You will want to use `scanf()` for this.

```
int num_players = 3;
printf("Number of players (3 to 11)? ");
int scanf_result = scanf("%d", &num_players);
```

   The variable `scanf_result` stores the return value of `scanf()`. `scanf()` returns the number of elements read successfully, which in this case would either be 0 or 1. If the return value is 0, we have not read successfully, and must throw an error. We must also validate that the input is a number in the required range. In the case that the user inputs anything other than a valid integer between 2 and 10 inclusive, print the following error to `stderr` informing them of improper program usage, then use the default value of 3 as the number of players:

```
if (scanf_result < 1 || num_players < 3 || num_players > MAX_PLAYERS) {
    fprintf(stderr, "Invalid number of players. Using 3 instead.\n");
    num_players = 3;
}
```

   (What is `stderr`? In UNIX, every process on creation has access to the following input/output (I/O) streams: `stdin`, `stdout`, and `stderr`. A running program is a process. `stdin`, or standard input, is the input stream in which data is sent to be read by a process. `stdout`, or standard output, is the output stream where data written by a process is written to. The last stream, `stderr`, or standard error, is an output stream like stdout, but is typically used for error messages.)

2. Prompt the user to input the random seed for this run of LCR.

[3]Think about how this structure can be translated to code!

```
unsigned seed = 4823;
printf("Random-number seed? ");
scanf_result = scanf("%u", &seed);
```

In the event that the user inputs anything other than a valid seed[4], print the following error to `stderr` informing them of improper program usage, and then use the default value of 4823 as the random seed[5]:

```
if (scanf_result < 1) {
    fprintf(stderr, "Invalid seed. Using 4823 instead.\n");
}
```

3. Set the random seed and make sure that each player starts off 3 chips. Note that it isn't made explicit how you keep track of each player's chips. There are, of course, many ways to go about this. You are encouraged to keep things simple, however.

4. Proceed around the circle starting from player 0. For each player:

   (a) Print out the name of the player currently rolling the dice.[6] An array of player names that you must use is provided in the header file `names.h`. Index 0 of this names array is the name of player 0, index 1 is the name of player 1, and so on and so forth. Print the name using `printf()`, not `fprintf()` (since the player name is not an error). Use the `printf()` format string `"%s:"`.

   (b) Roll all the dice[7] that the player gets (remember that it is not greater than 3, but also not more than the number of remaining chips). Update who has what chip and report the roll. You should use `printf()` format string `" ends her turn with %d\n"`. Remember that this is the step where you use the PRNG program `cse_random()` to generate random die rolls. However, also remember to make sure that each roll is in the range $0 \leq \texttt{roll} \leq 5$ (see Section 3.2).

   (c) After the chips are redistributed, check if there is a winner. There is a winner if there is only one player remaining with any chips.[8] Use the `printf()` format string `"%s won!\n"` if there is a winner.

   (d) If there is no winner, continue on to the next player. This is the player to the left (clockwise).

You will find starting files for your assignment in the **resources** repository. Copy them into your personal repository using the commands below, where `yourcruzid` is the name of your personal repository.

```
cd ~/s24/13s/resources
git pull
cd ..
cp -r resources/asgn2/* yourcruzid/asgn2
```

[4]With the knowledge that you have, it is difficult to check if the seed entered is too big or negative. Thus, you can assume that those inputs will never be given.

[5]There is another issue that is difficult to control that you may ignore. If a player inputs a non number for the number of players, `scanf()` will be unable to read a seed either, and the default seed will be used. This is expected behavior.

[6]If the next player has no dice to roll, do not print her name. Simply continue to the next player.

[7]In the real game, this should be happening simultaneously. Think about how rolling the dice one at a time could be problematic . . .

[8]Remember that this may not be the player who's turn just ended. Also remember that there must be a winner every game. Think about why!

6 Testing Your Program

To receive full credit, the output of your program must match the output of a reference program that we provide. You can test your program by comparing its output to the output of the reference program automatically.

Two reference programs (often called a "binaries") can be found in the course resources repository on git.ucsc.edu. One is called `lcr_x86`, and one is called `lcr_arm`. You can use one of them to check your program's functionality. The only thing that should not match the reference binary exactly is any error messages that you print.

As in Assignment 1, you will be writing tests. You will again be provided with a runner.sh file. This time, all your tests are expected to pass on your code and our code. You will still be writing tests in the `tests` folder. Each test will be a bash script that tests one aspect of your code. One test has been provided, as writing tests in for this assignment is a bit more complicated.

7 Deliverables

You will need to turn in the following source code and header files. Note: do not turn in the executable file `lcr`. In this class you never add an executable file to your repository. Add only source code. You can do this by running `make clean` before you make every commit.

1. `lcr.c`: This C source-code file contains your implementation of the game. It must include the definition of `main()` and any supporting functions that you write. You will start creating it with the C code that is described in boxes in Sections 3.1, 3.2, and 5 (and the line `#include <stdio.h>`).

2. `design.pdf`: This document must be a proper PDF that is well formatted and readable. This document must describe your program as a whole and the design process that you used to create it. It will also describe the results of the program. More information about it can be found on the template. design process for your program with enough detail such that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudocode.

3. The `tests` folder and its contents:

   The `tests` folder should contain a few tests for your program. They should all follow the form `test_something.sh`. They should run with `bash`, and test some part of the spec.

4. `Makefile`: Do not edit this file. We provide this file, but you will turn it in to ensure that we can compile your program. This file directs program compilation, building the program `lcr` from `lcr.c`. Read this file carefully, and try to understand what it does, as you may not receive one in future assignments.

5. `names.h`: Do not edit this file. We provide this file, but you will be turning it in to ensure that we can compile your program. This file contains the array of player names to be used in your implementation of the game. Do not change the names of the players. The automatic grading system is looking for exact output from your program, including the player names.

6. `cse13s_random.c`: Do not edit this file. We provide this file, but you will be turning it in to ensure that we can compile your program.

7. `cse13s_random.h`: Do not edit this file. We provide this file, but you will be turning it in to ensure that we can compile your program.

8. `runner.sh` Do not edit this file. This provided file will examine your tests folder. You will not need to run it directly, the Makefile will handle that.

8 Submission

Remember: add, commit, push, and submit your commit ID on Canvas!

Your assignment is turned in only after you have pushed and submitted the commit ID that you want graded on Canvas. "I forgot to push" and "I forgot to submit my commit ID" are not valid excuses. It is highly recommended to commit and push your changes often.

Remember that you have two deadlines: You will be turning in a draft of your design early, and then you will be turning in a final version of your design report along with your source code.

Be sure to format your code using the make format rule in the Makefile provided. This requires installing the clang-format package, if you have not already. Not formatting before submitting your code will result in reduced points. Your `.clang-format` file should already be located in the root directory of your git repository so that it can be used for future assignments also. If you find that it is not present, you may also copy the one from the resources repository into your personal repository.

9 Supplemental Readings

The C Programming Language by Kernighan & Ritchie.

- Chapters 2–4.
- Chapter 7, §7.2, §7.4, §7.6.
- `man 3 printf`

10 Revisions

Version 1 Original.

Version 2 Maximum number of players changed from 10 to 11 to match `MAX_PLAYERS` in `names.h`.