Assignment 1
LRC

by Jessie Srinivas
Based on assignments by Dr. Darrell Long CSE 13S, Winter 2024
Document version 1 (changes in Section 10)

Due Friday January 26, 2024, at 11:59 pm
Draft Due Wednesday January 24, 2024, at 11:59 pm

1 Introduction

We are going to simulate [1] a simplified version of the dice game Left, Right, and Center [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.

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 table

- If the player rolls a 1, 2, or 3, (this is called a "dot") the player will keep one chip.
- If the player rolls a 4, (this is called "left") the player will pass one chip to the left (clockwise).
- If the player rolls a 5, (this is called "center") the player will place one chip in the pot.
- If the player rolls a 6, (this is called "right") the player will pass one chip to the right (counter-clockwise).

After that turn, the next player will take their turn. The game continues until only one person 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.

[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 can also be found here `https://bicyclecards.com/how-to-play/left-center-right/`

3.1 Enumerating the Positions

In this game, we roll a six sided die, but some of the faces will end up doing the same thing. While it is possible to represent this as a number, and check what the roll is with 6 if statements, the die can be better represented with an abstraction. First, rolling a 1, 2, or 3 gives the same result: a dot. 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 a `enum` (enumerated) type. This allows us to create names 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 roll. 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. An example of how this could be used is 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. 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

- This array is defined with the `const` keyword. This means that it can not be modified after it is created. This is useful to make sure that you don't mess up the die.
- The type of this array is `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). After you create that pseudorandom number, you will have to constrain it to being a 6 sided die. 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 $0 \leq roll < 6$

You will use the PRNG defined by the standard C library, which is interfaced with the two functions `srandom()` and `random()`. You will need to `#include <stdlib.h>` in order to use these functions. The `srandom()` function is essential in order to make your program reproducible. `srandom()` sets the random seed, which effectively establishes the start point of the pseudorandom number sequence that is generated. This means, after calling `srandom()` with a seed, that the pseudorandom numbers that are generated by `random()` always appear in the same order.

Try out the following test program, `prng.c`:

```
#include <stdio.h>
#include <stdlib.h>

#define SEED 4823

int main(void) {
    for (int i = 0; i < 3; i += 1) {
        printf("Set the random seed to %d.\n", SEED);
        srandom(SEED);
        for (int j = 0; j < 5; j += 1) {
            printf(" - generated %lu\n", random());
        }
    }
    return 0;
}
```

The output of running the pseudorandom number generator example is given here so you can check if the numbers produced on your system match the ones produced by the system in which your program will be graded on.

```
./prng
Set the random seed.
 - generated 966111538
 - generated 529203374
 - generated 326976793
 - generated 779784191
 - generated 644163029
Set the random seed.
 - generated 966111538
 - generated 529203374
 - generated 326976793
 - generated 779784191
 - generated 644163029
Set the random seed.
 - generated 966111538
 - generated 529203374
 - generated 326976793
 - generated 779784191
 - generated 644163029
```

4 Your Tasks

You will be performing three main tasks:

1. Design your program and test cases
2. Write your program in the C programming language
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 LRC 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 `lrc.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 10)? ");
   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 and 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 > 10) {
       fprintf(stderr, "Invalid number of players. Using 3 instead.\n");
   }
   ```

   (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 LRC.

   ```
   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, print the following error to `stderr` informing them of improper program usage, and then use the default value of 4823 as the random seed:

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

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

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. An array of player names that you must use will be 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 that the player gets (Remember that it is not greater than 3, but also not more than the number of remaining chips). Report the roll and update who has what chip. You should use `printf()` format string `" ends her turn with %d\n"`. This is the step where you use the PRNG program `random()` to generate a random die roll. Make sure that it is in the range $0 \leq \text{roll} < 6$ (see Section 3.1).

   (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.[4] 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).

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.

The reference program (often called a "binary") can be found in the course resources repository on `git.ucsc.edu`. You should clone it to your virtual machine and use it to check your program's functionality. The only thing that should not match the reference binary exactly is any error messages that you print.

Like assignment 0, 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. Again, you will put your tests in a folder called `tests`, and each one 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 `lrc`. 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. `lrc.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.

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.

[4] 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!

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 `lrc` from `lrc.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. `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 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 per

9 Supplemental Readings

The C Programming Language by Kernighan & Ritchie.

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

10 Revisions

Version 1 Original.