ComS 327 Project Part 4

Solitaire: interactive terminal game

Spring 2020

1 Summary

The final part of the project is to build an interactive solitaire game that is played in a terminal window. You will need to use a library (either `ncurses` or `termbox`) that supports displaying characters to specific locations in a UNIX terminal, and with desired color attributes. Your `Makefile` must now build the executables for all previous parts of the project, along with a new executable named `game`. Your executable should take the following switches.

- `-f file`: Initialize the game settings based on `file`, an exchange-format file. You may ignore any moves specified in the input file.
- `-s seed`: Initialize the game by randomly shuffling the deck and dealing out cards, using the specified `seed` (a positive integer) to initialize the random number generator. Using the same seed should cause the same game to be dealt.
- `-1`: Specify `turn 1` if a random game is generated.
- `-3`: Specify `turn 3` if a random game is generated.
- `-l L`: Specify `limit L` if a random game is generated (default is "`unlimited`").

Note that you are not required to implement all of these (see Section 3). You may implement additional switches if you like (for example, to help with debugging, or to implement extra features). Implementation may be in C, C++, or a combination of the two.

2 Interface

The details of the user interface are not specified. However, your interface must contain the following elements.

- Information about which hotkeys are active and what they do (for example, if the user is expected to press "Q" to quit, then this should be clear from the interface).
- The input file name, or the random number generator seed, used to initialize the game.
- A display with the tableau, foundations, and top of the waste pile, that is updated as the game is played.

Your code will be tested on `pyrite`, in a terminal with at least 80 columns and at least 25 rows. Figure 1 is a screenshot of an example interface (built with `termbox`) that satisfies the requirements. For reference, the game shown in the Figure is from file `game.txt`:

```
RULES:
  turn 3
  limit 2
FOUNDATIONS:
```

```
  _c
  _d
  _h
  _s
TABLEAU:
  3h 7h 8h Th Jc Ad | Qs
  7c 6c 6s 7s 5h | Tc 9d 8c
  7d As | Ks Qh Js
  Kd 9s | 6h
  Td | Qc
  |
  | 4d 3c 2h
STOCK:
Jd 3d Ac | Kc 4h 2s Qd 6d Kh 5s 4s Ah Ts 4c 5d 2d 9c 9h 2c Jh 8s 3s 8d 5c
MOVES:
```

3 Features

The game must be playable when turning over 1 card, and with unlimited stock resets. You must be able to read from an input file, or generate a game from a random seed. All additional features are worth extra credit.

3.1 Turning over 3 cards

This means you must support `turn 3` in the input file, and/or implement the `-3` switch. Game play should turn over 3 cards at a time, and the waste top display should be able to show the top 3 waste cards (or fewer).

3.2 Limited stock resets

This means you must support `limit L` in the input file, and/or implement the `-l L` switch. Game play should disallow stock resets when the limit is reached, and the display should be updated to show the number of remaining stock resets.

3.3 Undo moves

There should be a hotkey that un-does the last move. You may support a single un-do (i.e., you only remember the previous game configuration) or, for more points, you may support unlimited (or a large, fixed number) of un-dos.

3.4 Shuffling

Shuffling a deck of cards may be done efficiently and perfectly (meaning, shuffling more than once will not produce a better shuffle than shuffling once) as follows. Use an array of 52 cards, in any initial order (the easiest is to initialize the cards in rank, suit order). Then, make a pass through the array, a bit like selection sort. To determine which element should be at position i, choose randomly from the elements from position i to the last element in the array, included. For instance, assuming the cards are represented by integers 1 through 52:

```
void shuffle(int* deck)
{
  int i;
  for (i=0; i<52; i++) deck[i] = i+1;
```

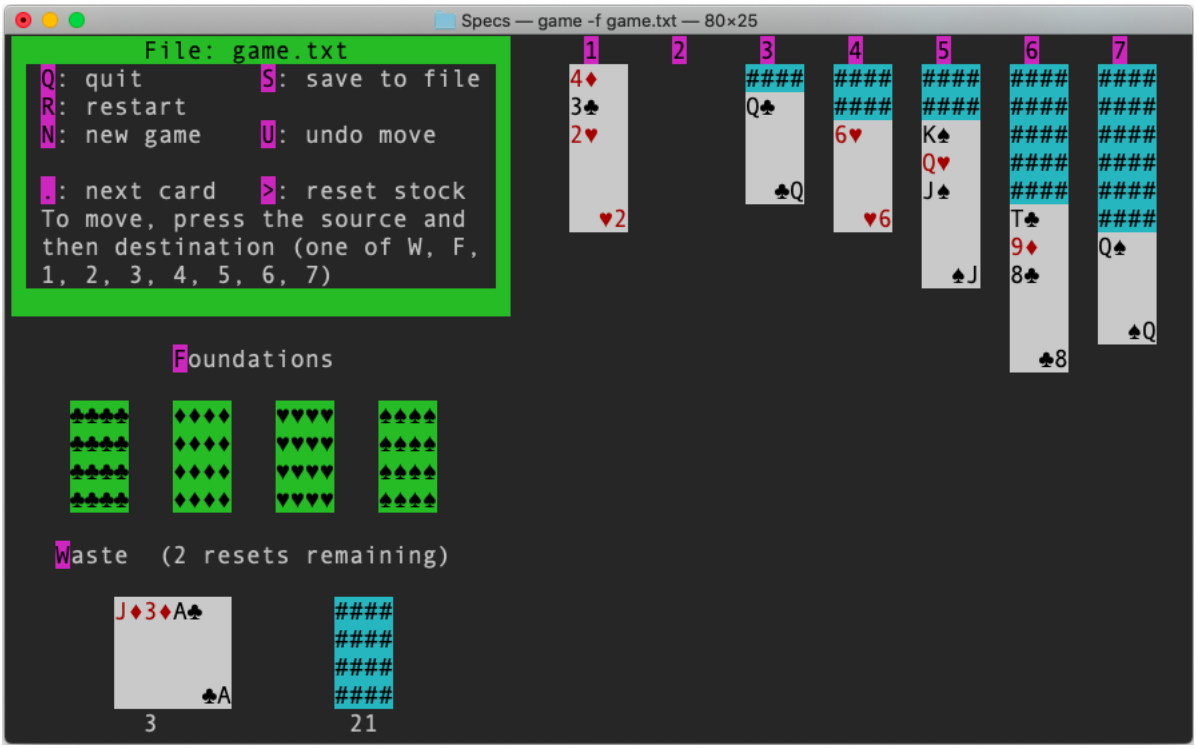

Figure 1: Screen shot of an example interface

```
  for (i=0; i<51; i++) {
    int j = choose_randomly_between(i, 51);
    if (i != j) {
      Swap(deck[i], deck[j]);
    }
  }
}
```

To choose "randomly" between (and including) integer values a and b, we need a random number generator that can produce real values randomly and uniformly strictly between 0 and 1. Assume we have a function `Random()` that does this. Then we could use:

```
long choose_randomly_between(long a, long b)
{
  return a + (long) ( (b-a+1) * Random() );
}
```

To implement `Random()`, you may use functions `random()` and `srandom()` from the standard C library, or you may implement your own. The following is a simple but effective Lehmer random number generator:

```
unsigned long RNG_seed;

double Random()
{
  const unsigned long MODULUS = 2147483647;
  const unsigned long MULTIPLIER = 48271;
  const unsigned long Q = MODULUS / MULTIPLIER;
  const unsigned long R = MODULUS % MULTIPLIER;
```

```
  unsigned long t1 = MULTIPLIER * (RNG_seed % Q);
  unsigned long t2 = R * (RNG_seed / Q);

  if (t1 > t2) {
    RNG_seed = t1 - t2;
  } else {
    RNG_seed = t1 + (MODULUS - t2);
  }

  return ((double) RNG_seed / MODULUS);
}
```

The global variable `RNG_seed` must be initialized to an integer greater than 0 and less than `MODULUS`.

4 Grading

The project is worth 100 points for individuals, and 110 points for groups of 2. However, there is ample opportunity for extra credit.

6 pts	`README` file, that describes implemented features. In particular, indicate which of the `-f` and `-s` switches are implemented, as **TAs will grade based on this**
7 pts	`DEVELOPERS` file
7 pts	Working `Makefile`
8 pts	Your `check` executable still works
8 pts	Your `advance` executable still works
12 pts	Tableau display
8 pts	Foundations display
8 pts	Waste top display
8 pts	Hotkeys display
14 pts	Game play
4 pts	Quit exits cleanly
10 pts	`-f` switch works
10 pts	`-s` switch works
5 pts	Turn 3
5 pts	Limited stock resets
5 pts	Undo last move
5 pts	Undo several moves

5 What to submit

5.1 In your git repository

Remember to include the following in your git repository.

- All TAs and the instructor as "reporters" (with read access).
- C and/or C++ Source code.
- A `Makefile`, so your executables can be built by simply typing "`make`". The TAs will **build and test your code on `pyrite`**.
- A `README` file, to indicate which features are implemented. The TAs will grade the `-f` and `-s` switches based on what your `README` file says are implemented.

- A `DEVELOPERS` file, with necessary information about the source code for other developers. For group submissions, this file also should indicate which student(s) implemented which function(s).
- A tag with an appropriate name (e.g., "Part4", or "Part4a", "Part4b" in case you need to create more than one tag) as a frozen snapshot for the TAs to grade.

5.2 In Canvas

Please submit under the appropriate assignment in Canvas: either as an individual submission (you worked by yourself) or as a group submission (you worked in a group of 2 students). In the text submission box, indicate:

- The webpage URL for your (group's) git repository. Please indicate if this is a different repository than the one(s) you used previously.
- The name of the git tag for the TAs to grade.
- For group submissions, indicate which students (names and ISU netIDs) are part of the group.
- For group submissions, *all* students in the group should submit this information in Canvas.