ComS 327 Project Part 2

Solitaire: process moves

Spring 2020

1 Summary

For the second part of the project, you must read an input file that describes a game configuration (possibly in the middle of a game) followed by a sequence of zero or more moves. If the input file is valid, then you must check that the sequence of moves is valid, and display the game state after the moves. For this part of the project, all source code must be C.

Specifically, your `Makefile` must build your `check` executable from part 1, and a new executable named `advance`. Program `advance` should accept the following optional command-line switches and arguments, which may appear any number of times and in any order. If two or more switches conflict with each other, the last switch takes precedence. You may implement additional switches, for example to display debugging information, if you wish.

- Switch `-m N`, indicating that at most N moves should be played from the input file. If omitted, the default value of N is infinity (i.e., play *all* moves from the input file).
- Switch `-o file`, indicating that the game configuration output should be written to the specified file. If omitted, the game configuration output should be written to standard output. This switch is extra credit.
- Switch `-x`, indicating that the game configuration output should be in "exchange" format, i.e., in the same format as the input files. If this is omitted, then the game configuration output should be in "human readable" form (described below). This switch is extra credit.

At most one filename argument may be passed, in any position. When testing your executable, we will not use filenames that begin with a `'-'` character. If no filename is given, the program should read from standard input (just like program `check` in part 1).

2 Movement rules

Note that in Solitaire, the card ranks are Ace (lowest), 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King (highest), and the hearts and diamond suits are colored red, while clubs and spades are colored black. The following moves are allowed.

- **Moving the top waste card to the foundations**. If the top waste card is an Ace, it may be moved to the appropriate (empty) foundation. Otherwise, the top waste card may be placed on top of the card with one lower rank, in the same suit, in the foundations.
- **Moving the top waste card to the tableau**. If the top waste card is a King, it may be moved to an empty tableau column. Otherwise, the top waste card may be placed onto the bottom card in a tableau column, if the waste card has rank one lower than, and the opposite color of, the bottom card.
- **Moving a tableau card to the foundations**. If the bottom card of a tableau column is an Ace, it may be moved to the appropriate (empty) foundation. Otherwise, the bottom card of a tableau column may be placed on top of the card with one lower rank, in the same suit, in the foundations.

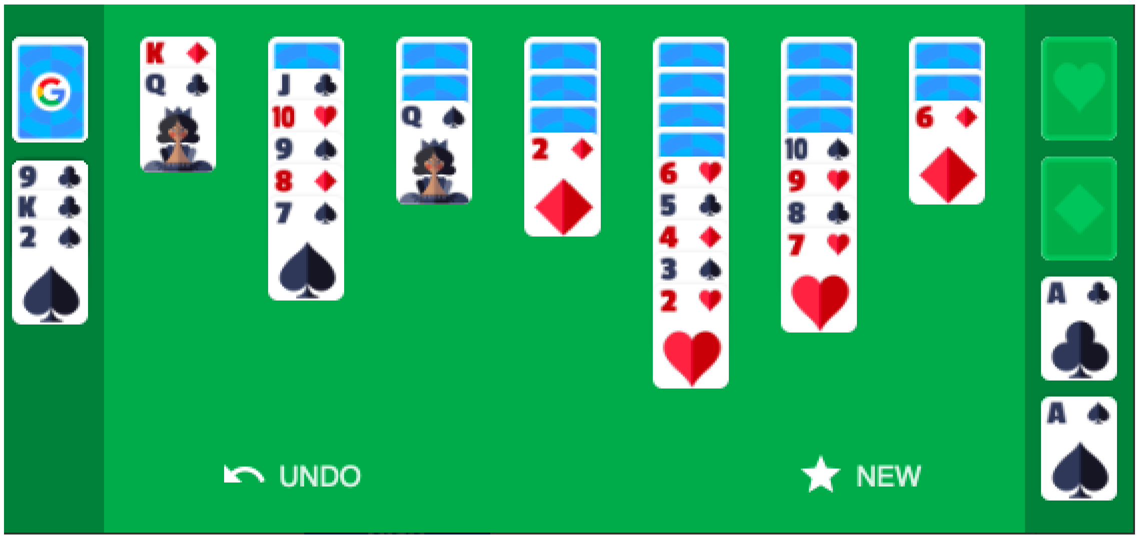

Figure 1: Example to illustrate movement rules

- **Moving tableau cards from one column to another**. Any pile of cards, starting from any uncovered card, and including all cards down to the bottom of the column, may be moved from one tableau column to another column, if the top card in the pile could be moved to the destination tableau column as a waste card.
- **Turning over the next cards**. If there are cards remaining in the stock, we may turn over (either 1 or 3, depending on the rules) stock cards and place them on top of the waste pile.
- **Resetting the stock pile**. If there are no cards remaining in the stock, we may turn over the waste pile and use it as the stock again. The first card added to the waste pile (on the bottom) becomes the first card of the stock pile, and the last card placed on the waste pile becomes the last card of the stock pile (i.e., we go through the stock again, in the same order, except for any cards that were moved). Different variations of Solitaire have different rules about the number of stock resets allowed.

If moving cards from a tableau column causes the bottom card of the column to be a hidden (covered) card, the card is immediately uncovered. Thus, there is never a tableau column whose bottom card is hidden.

As an example, for the game shown in Figure 1, there are several possible moves:

1. We could move the `2s` from the top of the waste pile to the foundations.
2. We could move the `6d` from column `7` on top of the `7s` in column `2`. We would then uncover the bottom card of column `7`.
3. We could move the pile from `6h` down to `2h` in column 5 on top of the `7s` in column 2. We would then uncover the bottom card of column 5.
4. We could move the pile from `5c` down to `2h` in column 5 on top of the `6d` in column 7.
5. We could turn over the next card(s) from the stock to the waste.

3 Input file format

The input file format is the same as for part 1, except you will need to parse and process the `MOVES:` section. As a reminder (see part 1 specs for details), the input file has the following sections, always in this order.

1. The `RULES:` section indicates which variant of Solitaire is being played, and contains:

- “`turn` T”, where T will be 1 or 3 and indicates how many cards should be turned over at a time from the stock to the waste.
- “`unlimited`” or “`limit` R” where R indicates the number of times that the waste can be “reset” back into the stock. Keyword “`unlimited`” specifies $R = \infty$.

2. The `FOUNDATIONS:` section indicates the current top card of each of the four foundations.
3. The `TABLEAU:` section indicates the current status of the seven tableau columns.
4. The `STOCK:` section indicates both the waste and the remaining stock.
5. The `MOVES:` section contains zero or more moves. Each move is separated by an arbitrary amount of whitespace and possibly comments. A move is one of:
   - “`.`” causing T cards to be turned over from the stock to the waste.
   - “`r`” causing the waste pile to be reset back to the stock.
   - *src*`->`*dest*, where *src* and *dest* are single characters. The *src* is either `'w'` for the top of the waste, or a column number `'1'`, `'2'`, ..., `'7'`. The *dest* is either `'f'` for the foundations, or a column number `'1'`, `'2'`, ..., `'7'`. This indicates to move a card or pile of cards (as appropriate) from the source location to the destination location. For example, “`w->4`” says to move the top card from the waste pile to column 4. Note that there cannot be any whitespace characters between *src* and “`->`”, or between “`->`” and *dest*.

   The list of moves indicates a *sequence* of moves, played one at a time, to advance the state of the game.
6. The end of the file. This is the only indication that the previous section has ended.

Example input files are given in Section 7.

4 What to check for

4.1 Formatting errors

You should continue to check that input files are valid (ideally you are just extending the parser you wrote for part 1). As with part 1, formatting errors in the input file should generate an appropriate error message that includes the line number of the input file where the error occurs. In addition to the checks from part 1, you should check for formatting errors in the `MOVES:` section. Any moves not of the form specified above would be considered a formatting error. This includes incorrect source or destination characters.

4.2 Invalid moves

You will also check that the moves are valid for the current game state. Note that this is *not* considered to be a formatting error. For example, moves “`4->f`”, “`.`”, and “`r`” are correctly formatted, but might be invalid: if in the current game state, the card at the bottom of column 4 cannot be moved to the foundations, then “`4->f`” is an invalid move. Similarly, if there are no remaining cards in the stock, then “`.`” is an invalid move. Finally, if the stock is not empty, or there are no stock resets left, then “`r`” is an invalid move.

Moves are applied sequentially, starting from the initial game configuration. If there are no moves, then the final game configuration is the same as the initial game configuration. Otherwise, the first move m_1 is taken, which leads to a new game configuration, then the second move m_2 is taken, which leads to another game configuration, and so on:

$$\text{initial} \xrightarrow{m_1} \text{config } 1 \xrightarrow{m_2} \text{config } 2 \xrightarrow{m_3} \cdots \xrightarrow{m_n} \text{config } n$$

This continues until either all moves have been processed, or an invalid move is discovered.

5 Output

If all processed moves are valid, then your program should write, to standard output:

```
Processed N moves, all valid
```

where N is the total number of moves played. This may be fewer than the number of moves in the input file, if the `-m` switch is given. If not all moves are valid, then your program should write, to standard output:

```
Move M is illegal: (move)
```

where M is the number (counting from 1) of the first invalid move, and `(move)` indicates the move, using the same format for moves as given in the `MOVES:` section of the input file.

Then, your program should display (or write to a file, depending on the command-line switches) the current game state configuration. If all N moves are valid, then display the game state after the N moves are taken. If move M is invalid, then display the game state immediately before the M^{th} move.

5.1 Exchange format

If the `-x` switch is given, output the game state using the input file format. This means, any of the project utilities that read from an input file, should be able to read this output file. Your output file should contain no moves in the `MOVES:` section.

5.2 Human-readable format

If the `-x` switch is not given, then the output should be a "human-readable" format, which is exactly as follows (see examples in Section 7).

- The keyword "Foundations", then a newline.
- The top foundation card for each suit, separated by exactly one space. Suits should be in order: clubs, diamonds, hearts, spades. If a foundation is empty, display a rank of `'_'`. Then a newline.
- The keyword "Tableau", then a newline.
- Seven columns, each separated by exactly one space. For hidden cards, display `"##"`. For missing cards, display `".."`. Display as many rows as needed, one per line, up until but not including the first line containing `".."` in each column.
- The keywords "Waste top", then a newline.
- The top card(s) of the waste pile. If the waste pile is empty, display `"(empty)"`. Otherwise, display the top T (or as many as possible, if fewer than T) cards of the waste pile, separated by exactly one space. Then a newline.

6 Grading

The project is worth 100 points for individuals, and 110 points for groups of 2. You must handle the following `RULES:`

- `turn 1` (turn over one card at a time)
- `unlimited` (no limits to resetting the stock pile)

Additional rules are worth extra credit. The points breakdown for various features of the project are listed below. You may implement more features than required, for extra credit.

10 pts	`README` file, that describes implemented features
10 pts	`DEVELOPERS` file, that gives an overview of your implementation, including a breakdown of source files and functions, and who authored each function.
10 pts	Working `Makefile`. Typing “`make`” should build executables `check` and `advance`.
6 pts	Reads from `stdin` if no filename argument
6 pts	Reads from filename passed as argument
3 pts	Summary output to `stdout`
8 pts	Output is in human-readable format
12 pts	Correctly processes moves
8 pts	`-m` switch
10 pts	Appropriate error messages for formatting errors
12 pts	Catches invalid moves
5 pts	Stress tests
5 pts	Works for `turn 3`
5 pts	Works for `limit R`
10 pts	`-x` switch for exchange format output
5 pts	`-o` switch

7 Examples

7.1 File `testmoves1.txt`

```
#
# Start of a game
#
RULES:
  turn 1
  unlimited
FOUNDATIONS:
  _c
  _d
  _h
  _s
TABLEAU:
  3h 7h 8h Th Jc Ad | Qs
  7c 6c 6s 7s 5h | Tc
  7d As Ks 8c | Js
  Kd 9s 6h | 3c
  Td Qc | 9d
  2h | Qh
  | 4d
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:
#
# Lots of moves
#
  4->1  5->2  5->3  2->5  2->1  5->2  5->f  5->3  .     w->7
  6->7  .     .     w->f  .     .     .     .     .     .
  .     w->4  1->4  w->1  7->1  7->f  3->1  .     w->6  .
```

```
w->f  .     .     .     .     w->f  .     .     .     w->f
.     w->3  .     .     .     .     r     .     w->f  .
w->5  .     .     w->f  4->f  4->f  .     w->5  7->5  7->5
.     .     w->3  .     w->f  .     .     w->5  7->5  .
w->3  .     w->3  7->3  7->f  .     w->f  6->f  .     .
w->f  r     .     w->f  6->f  6->5  6->3  6->1  4->6
```

7.1.1 Output example 1

If we run

```
./advance -m 999 testmoves1.txt
```

then the output should be

```
Processed 89 moves, all valid
Foundations
5c 3d 5h 4s
Tableau
Kh Ks ## ## Kc 7c ..
Qs Qh Qc 9s Qd 6h ..
Jd Js Jh .. Jc 5s ..
Tc .. Ts .. Th 4d ..
9d .. 9h .. 9c .. ..
8c .. 8s .. 8h .. ..
7d .. 7h .. 7s .. ..
6c .. 6s .. .. .. ..
Waste top
(empty)
```

7.1.2 Output example 2

If we run

```
./advance -m 15 testmoves1.txt
```

then the output should be

```
Processed 15 moves, all valid
Foundations
Ac _d _h As
Tableau
4d Ks ## ## .. ## ##
3c Qh ## ## .. ## ##
2h Js 9d 6h .. ## ##
.. .. 8c .. .. ## ##
.. .. 7d .. .. 5h ##
.. .. .. .. .. .. ##
.. .. .. .. .. .. Qs
.. .. .. .. .. .. Jd
.. .. .. .. .. .. Tc
Waste top
Kc
```

7.1.3 Output example 3

If we run

```
./advance -m 15 testmoves1.txt -o foo.txt
```

which contains the extra credit switch "-o", then the output should be

```
Processed 15 moves, all valid
```

and file `foo.txt` should contain

```
Foundations
Ac _d _h As
Tableau
4d Ks ## ## .. ## ##
3c Qh ## ## .. ## ##
2h Js 9d 6h .. ## ##
.. .. 8c .. .. ## ##
.. .. 7d .. .. 5h ##
.. .. .. .. .. .. ##
.. .. .. .. .. .. Qs
.. .. .. .. .. .. Jd
.. .. .. .. .. .. Tc
Waste top
Kc
```

7.2 File `testmoves2.txt`

```
RULES: turn 1 unlimited
FOUNDATIONS: Ac 3d 3h 3s
TABLEAU:
  4c 5h | Ts
  2c | 7s 6d
  | Kc Qd Js Td 9s
  6s 8h Qc | 7c 6h 5s 4h 3c
  8d | Qs Jd Tc 9d 8c 7d 6c 5d 4s
  |
  | Ks Qh Jc Th 9c
STOCK:
5c 4d Kh Jh 9h 7h Kd 8s |
MOVES:
  r
  . w->6 . w->6 . w->2 3->2 3->5 4->5
# This move is invalid:
  4->3
```

7.2.1 Output example 4

If we run

```
./advance < testmoves2.txt
```

then the output should be

```
Move 11 is illegal: 4->3
Foundations
Ac 3d 3h 3s
Tableau
Ks Kh .. ## Kc ## ##
Qh Qs .. ## Qd 7s ##
```

```
Jc Jd .. Qc Js 6d Ts
Th Tc .. .. Td 5c ..
9c 9d .. .. 9s 4d ..
.. 8c .. .. 8d .. ..
.. 7d .. .. 7c .. ..
.. 6c .. .. 6h .. ..
.. 5d .. .. 5s .. ..
.. 4s .. .. 4h .. ..
.. .. .. .. 3c .. ..
Waste top
(empty)
```

7.2.2 Output example 5

If we run

```
cat testmoves2.txt | ./advance -m 0
```

then the output should be

```
Processed 0 moves, all valid
Foundations
Ac 3d 3h 3s
Tableau
Ks .. ## ## Kc ## ##
Qh .. Qs ## Qd 7s ##
Jc .. Jd ## Js 6d Ts
Th .. Tc 7c Td .. ..
9c .. 9d 6h 9s .. ..
.. .. 8c 5s .. .. ..
.. .. 7d 4h .. .. ..
.. .. 6c 3c .. .. ..
.. .. 5d .. .. .. ..
.. .. 4s .. .. .. ..
Waste top
8s
```

7.2.3 Output example 6

If we run

```
./advance testmoves2.txt -m 1 -x -o out.txt
```

which contains the extra credit switches "`-o`" and "`-x`", then the output should be

```
Processed 1 moves, all valid
```

and file `out.txt` could contain

```
RULES:
  turn 1
  unlimited
FOUNDATIONS:
  Ac
  3d
  3h
  3s
```

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

8 What to submit

8.1 In your git repository

Remember to include the following in your git repository.

- All TAs and the instructor as "reporters" (with read access).
- C Source code.
- A `Makefile`, so your executable 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.
- 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., "Part2", or "Part2a", "Part2b" in case you need to create more than one tag) as a frozen snapshot for the TAs to grade.

8.2 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 you used for part 1.
- The name of the git tag for the TAs to grade.
- For group submissions, indicate which students are part of the group.