02322 Machine Oriented programming

Project 2

V3

Card game implemented using linked lists

Project 2 (just like project 1) is a deliverable in this course and is going to contribute to your grade for the course. The course is evaluated as a whole.

In this project we are going to implement in C (using linked lists) a card game called Yukon shown in Figure 1. You'll implement the game using text displayed in the terminal window, but you'll also need to create a graphical user interface for the game like the one shown in Figure 1. The game will be implemented in steps where you'll gradually implement the game, starting with functionality such as shuffling the cards and ending up with a full implementation of the game. In case you want to try out the game and get familiar with the game rules and gameplay, you can see an online implementation of the game here: https://www.icardgames.com/yukon.html

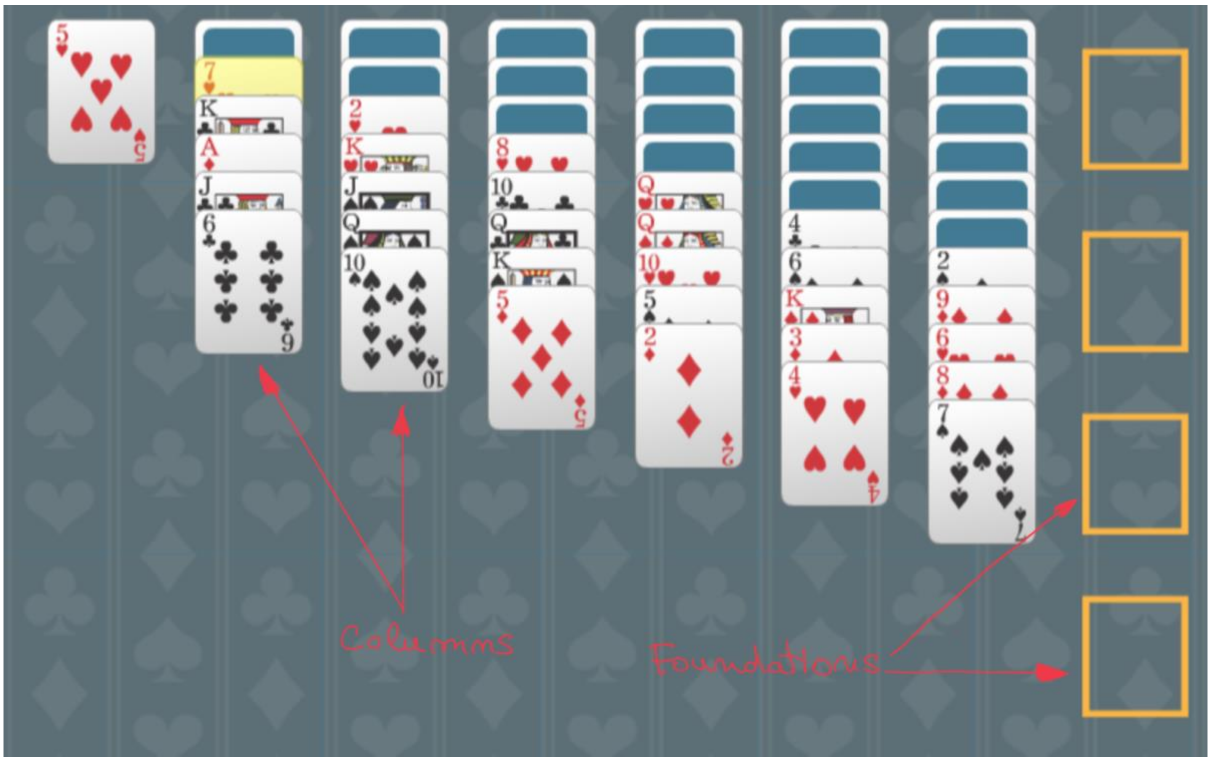


Figure 1 Layout of cards at the start of the Yukon game

Here is a short description of the rules for this game:

To play Yukon Solitaire, the game begins by dealing the entire deck into 7 columns. Starting from the left, the columns contain 1, 6, 7, 8, 9, 10, and 11 cards respectively. Out of these, 21 cards are dealt face down, with the remaining cards dealt face up (see Figure 1 for layout). The objective is to move all cards to the four foundation piles, organizing them by suit in ascending order from Ace to King.

Movement of cards within the tableau (columns) is distinctive. A card, along with any cards stacked beneath it, can be moved to another column if the card being moved is one rank lower and of a different suit than the bottom card of the target column. Unlike some other solitaire variations, the card moved does not need to be of alternating color, but it cannot be of the same suit.

If moving a card reveals a face-down card, that card is turned face up. The game is won when all cards are successfully moved to the foundation piles

Requirements

The cards from a deck of cards and from columns are stored in linked lists. You are not supposed to use arrays to store the value of the cards.

Start by developing a text-only version of the game, interacting through the terminal window. Once complete, retain the core game logic but transition the user interface from text-based to a graphical user interface (GUI). This results in two distinct versions of the game: a text-based implementation and a GUI-based implementation.

For displaying the cards as text on the terminal window, each card is represented with the help of 2 characters:

- First character represents the rank (value) of the card:
  A – Ace, numbered cards from 2 up to 9, T - 10, J – Jack, Q – Queen, K-King
- The second character represents the suit of the card:
  C - clubs (♣), D - diamonds (♦), H - hearts (♥), S – spades (♠)
- If the card is not visible (face down), then we just use parenthesis: [ ]

Here it's shown how the cards from Figure 1 should be represented on the terminal window:

```
C1      C2      C3      C4      C5      C6      C7

5H      []      []      []      []      []      []                      []      F1
        7H      []      []      []      []      []
        KC      2H      []      []      []      []                      []      F2
        AD      KH      8H      []      []      []
        JC      JS      TC      QH      []      []                      []      F3
        6C      QS      QC      QD      4C      []
                TS      KS      TH      6S      2S                      []      F4
                        5D      5S      KD      9D
                                2D      3D      6H
                                        4H      8D
                                                7S
LAST Command:
Message:
INPUT >
```

Figure 2 View of the game in the terminal window

Hint: Use tabs to separate the columns. For example: `printf("\tKC\t2H\n")`

Before displaying the cards, we first have a row with the name of the columns: C1 – C7. The foundations are also labeled with names from F1 to F4.

At the bottom, after all the columns are displayed there should be 3 lines containing the following:

- **Last Command:** Displays the last command that was inserted at the input prompt.
- **Message:** The message we've got from running the last command. It could be an error message if there was a problem running the last command or **OK** if the command was executed without error.
- **Input >:** a prompt called where we can input new commands.

In the start of the game there is no deck of cards loaded so we’ll have the following view:

```
C1    C2    C3    C4    C5    C6    C7

                                                []    F1

                                                []    F2

                                                []    F3

                                                []    F4
LAST Command:
Message:
INPUT >
```

Figure 3 Initial view

Here are the commands that your game should support:

STARTUP phase

First, we look at commands that are used before starting the game (let’s call it STARTUP phase) which allow us to get a deck of cards, shuffle it, start playing the game and quitting the program.

1. LD <filename>

This command loads a deck of cards with all cards initially hidden, represented as '[ ]' in the interface. The Figure 4 illustrates this starting arrangement.

```
C1    C2    C3    C4    C5    C6    C7

[]    []    []    []    []    []    []            []    F1
[]    []    []    []    []    []    []
[]    []    []    []    []    []    []            []    F2
[]    []    []    []    []    []    []
[]    []    []    []    []    []    []            []    F3
[]    []    []    []    []    []    []
[]    []    []    []    []    []    []            []    F4
[]    []    []

LAST Command:LD
Message:OK
INPUT >
```

Figure 4 Game view after a new deck of cards was loaded (value of the cards is hidden)

- **If <filename> is provided:** The command attempts to load the deck from the specified file. The file should contain one card per line, like this:

5D
5S
KD
9D
...

Each card is represented by its value and suit (e.g., 5D for Five of Diamonds). The program checks the file for correctness, ensuring each card is valid and the deck comprises exactly 52 cards, with 13 cards in each of the four suits (Clubs, Diamonds, Hearts, Spades) from Ace (A) to King (K). If the file is valid, the message **OK** is returned. If the file is invalid, an error message details the first encountered issue, including the line number.

- **If <filename> is not provided:** A new, unshuffled deck is loaded by default, starting with all Clubs from A to K, followed by Diamonds, Hearts, and Spades in that order. In this case, the command simply returns **OK**.
- **If the specified filename does not exist:** An error message indicates that the file does not exist.

This command ensures either the successful loading of a specified deck or the creation of a new, standard deck, with clear feedback provided in each scenario.

2. SW

This command displays all the cards in the terminal, arranged in the current order within the deck. Kind of like unhiding the cards by turning them face up. For example, if the deck is unshuffled, the display will appear as shown in Figure 5, with cards in sequential order starting from the Ace of Clubs. If no deck is loaded, the system returns an error message. If a deck is successfully displayed, the system responds with **OK**.

```
C1     C2     C3     C4     C5     C6     C7

AC     2C     3C     4C     5C     6C     7C             []     F1
8C     9C     TC     JC     QC     KC     AD
2D     3D     4D     5D     6D     7D     8D             []     F2
9D     TD     JD     QD     KD     AH     2H
3H     4H     5H     6H     7H     8H     9H             []     F3
TH     JH     QH     KH     AS     2S     3S
4S     5S     6S     7S     8S     9S     TS             []     F4
JS     QS     KS

LAST Command:SW
Message:OK
INPUT >
```

Figure 5 Showing an unshuffled deck of cards

3. SI <split>

This command shuffles the deck by interleaving cards from two separate piles. The deck is first divided into two piles:

- **If <split> is provided:** The top <split> number of cards form the first pile, and the remaining cards form the second pile. <split> must be a positive integer and less than the total number of cards in the deck.
- **If <split> is not provided:** A random number, less than the total number of cards, is used to divide the deck into two piles.

After splitting, we combine the piles to form a new 'shuffled pile' by alternately placing the top card from each pile until one pile is exhausted. The rest of the cards from the remaining pile are then placed at the bottom of the shuffled pile. This shuffled pile now becomes the current deck.

4. SR

This command performs a random shuffle of the deck. Imagine the current deck as the 'unshuffled pile.' During the shuffle, we take the top card from the unshuffled pile and insert it at a random position within a new pile, which we will refer to as the 'shuffled pile.' We repeat this process until there are no cards left in the unshuffled pile. The shuffled pile is then considered the current deck.

Note: The process begins with the shuffled pile being empty, so the first card has only one position available. With each new card, the possible positions increase by one—either at the top, at the bottom, or between any two existing cards in the pile. This method ensures each card has an equal chance of being placed in any position, resulting in a randomly shuffled deck.

5. SD <filename>

This command saves the current card deck to a file. If <filename> is provided, the deck is saved under that name. If no filename is specified, the default filename "cards.txt" is used. The file will list the cards one per line, following this format:

```
5D
5S
KD
```

....

Each card is denoted by its rank followed by its suit.

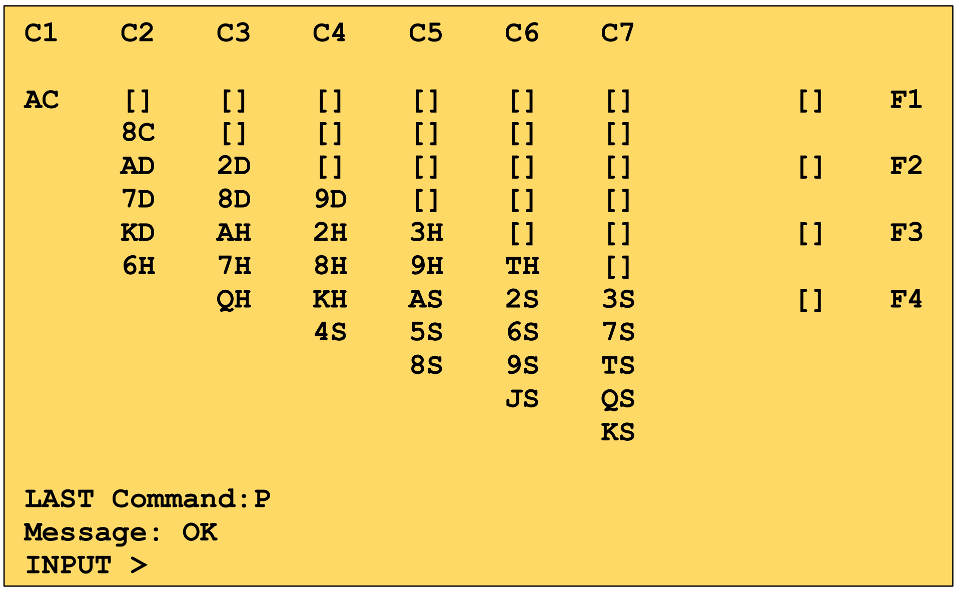
6. QQ

Command that quits the program. The program exits.

PLAY phase

7. P

Initiate a game with the current card deck by using this command, which transitions the game into the PLAY phase. Cards are dealt from the leftmost column (C1) to the rightmost (C7), filling one row at a time. Once in the PLAY phase, the commands specific to the STARTUP phase are no longer available. If such a command is attempted, the user will receive an error message stating, “Command not available in the PLAY phase.” Figure 6 illustrates the initial game setup using an unshuffled deck, as seen with the deck arrangement in Figure 5.



```
C1    C2    C3    C4    C5    C6    C7

AC    []    []    []    []    []    []          []    F1
      8C    []    []    []    []    []
      AD    2D    []    []    []    []          []    F2
      7D    8D    9D    []    []    []
      KD    AH    2H    3H    []    []          []    F3
      6H    7H    8H    9H    TH    []
            QH    KH    AS    2S    3S          []    F4
                  4S    5S    6S    7S
                        8S    9S    TS
                              JS    QS
                                    KS

LAST Command:P
Message: OK
INPUT >
```

Figure 6 Cards at the start of the game if an unshuffled card deck is used

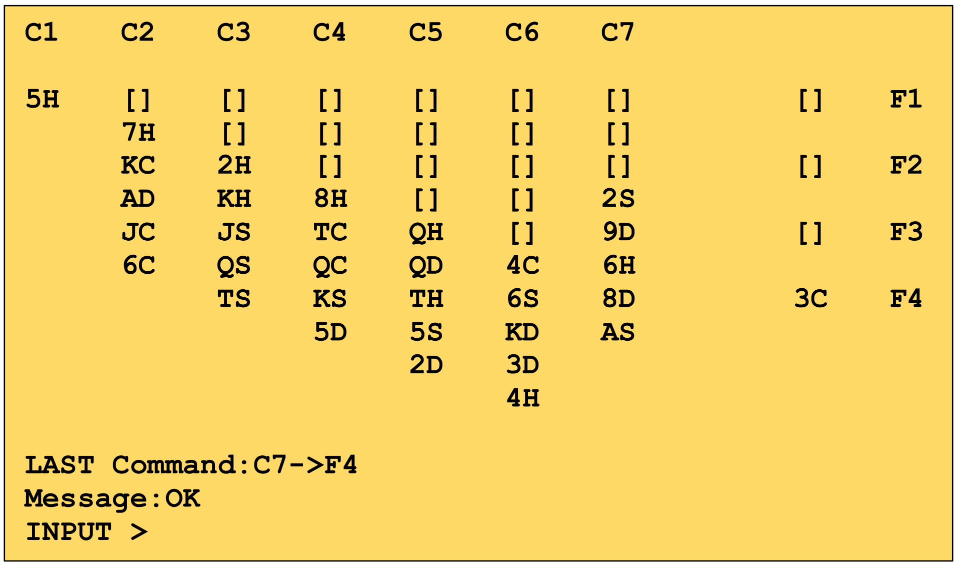
8. Q

Command that quits the current game and goes back to the STARTUP phase. The memory still contains the deck of cards used to play the game that we are quitting. So, if we use the command P again after Q, we basically restart the last game.

9. <Game Moves>

Move commands follow the format: <from> -> <to>. Here's how to specify each:

- **<from>:** The source card or pile:
  - A specific card in a column: <column>:<card> (e.g., 'C6:4H' for the 4 of Hearts in column 6).
  - The bottom card in a column: <column> alone indicates the bottom card (e.g., 'C6').
  - The top card from a foundation: Simply use the foundation number (e.g., 'F3').
- **<to>:** The destination:
  - To a column's bottom: <column> (e.g., 'C4').
  - To the top of a foundation: <foundation> (e.g., 'F2').



```
C1    C2    C3    C4    C5    C6    C7

5H    []    []    []    []    []    []          []    F1
      7H    []    []    []    []    []
      KC    2H    []    []    []    []          []    F2
      AD    KH    8H    []    []    2S
      JC    JS    TC    QH    []    9D          []    F3
      6C    QS    QC    QD    4C    6H
            TS    KS    TH    6S    8D          3C    F4
                  5D    5S    KD    AS
                        2D    3D
                              4H

LAST Command:C7->F4
Message:OK
INPUT >
```

Figure 7 Game view while playing

Using Figure 7 for reference, here are some move examples:

'C6:4H->C4' moves the 4 of Hearts from column 6 to the bottom of column 4.
'C6:4C->C4' moves the 4 of Clubs and all cards below it from column 6 to the bottom of column 4.
'C7:AS->F2' moves the Ace of Spades to foundation 2.
'F4->C6' moves the top card of foundation 4 (the 3 of Clubs) to column 6.

Validity Rules:

- The card moved must exist at the specified location.
- Moving a card to a column is only valid if the bottom card of that column is one rank higher and of a different suit.
- A card can only be added to a foundation from the bottom of a column and must follow suit and be one rank higher than the top card of the foundation.

- Only the top card from a foundation can be moved to a column, and it must follow the rank and suit restrictions for columns.

If a move is valid, the system returns **OK**. Otherwise, an error message states that the move is not valid.

10. GUI

Develop a graphical user interface (GUI) for the game. You can choose from several approaches:

- **Using C and SDL:** For those who wish to implement the GUI with C, the SDL library is an excellent tool. Begin with these resources:
  - Watch this introductory video on SDL with C: https://youtu.be/yFLa3ln16w0
  - Visit the SDL official website: http://libsdl.org
  - Explore SDL tutorials: https://lazyfoo.net/tutorials/SDL/
- **Using a different programming language:** Alternatively, you may create the GUI using a programming language of your choice (Java, Python, C#, etc.). Your existing C implementation will serve as the backend, handling the game's logic. The frontend (GUI) will communicate with the backend through TCP/IP sockets.

Choose the method that best suits your skills and project requirements.

Extensions

If the time allows it, you are welcome to implement one or more of the following extensions for extra credit:

- Validate the input. Make the program robust enough to handle any kind of input you use for the INPUT prompt. The program should not crash even if you write incorrect commands. In that case it should just return error messages.

- Implement extra instructions:

1. U

Undo the last move. You should be able to undo as many moves as you want until the start of the game.

2. R

Redo the previous move. It only works if you've run an undo command before.

3. S <filename>

Save the current state of the game to a file specified by parameter <filename>. It's up to you to choose the file format of the file.

4. L <filename>

Load a game from a file. Restores the game to a previous state that it had when it was saved with the S command.