VG101 — Introduction to Computer and Programming

Project 2

Manuel — UM-JI (Summer 2019)

- Include simple comments in the code
- Split the code over several functions
- Extensively test your code and impove it
- Start early and respect the milestones
- Update the `README` file for each milestone
- Update the `Changelog` file between two milestones
- Archive the files (`*.{zip|tar}`) and upload on Canvas

1 Project Setup

After successfully taking their very easy MATLAB midterm Haruka, Kana, and Chiaki, want to know more about programming. So they are thinking of the best way to learn and practice C, but would like it to be fun. Keeping this idea in mind, Haruka suggests to implement their favorite card game, *One Card*, a very simple shedding game, that is a game where the goal is to be the first one to get rid of all his cards.

The three sisters are pretty happy with this idea, and Kana feels this is a perfect opportunity to really practice good quality coding, and better understand why and how to best organise a large programs.

Overview

One Card is a rather simple game payed by n persons over a pre-decided number of rounds r. Two decks of Poker cards, excluding Jokers, are shuffled and c cards are offered to each player. Once everybody has received his cards, the dealer poses a card, face up in front of each player. The game will be played counter-clockwise, starting with the player who received the card with lowest rank. Once the playing order has been decided all those initial n cards are directly put in the discard pile. Finally the first card of the stock pile is laid face-up to initiate the rank and suit.

As the game starts each player, following the defined order, plays exactly one card either following the rank or the suit defined by the previous card. Any played card directly goes into the discard pile, and anyone who is unable to play should draw a card from the top of the stock pile. If the stock pile is exhausted, the discard pile is shuffled and used as stock pile.

As soon as a player has discarded all his cards the rounds stops. All other players receive a penalty equal to the number of cards left in their hands. The player who won the round initiates the following one, all the other rules remain unchanged. At the end of the r rounds the final score of each player is determined by summing up of all his penalties. The person with highest score wins. In case of equality more than one player can be declared winner.

Cards

Cards split into four main categories:

- Attack:
  - Cards with rank 2: the next player draws two cards from the stock pile;
  - Cards with rank 3: the next player draws three cards from the stock pile;
- Defense:
  - Cards with rank 7: cancel an attack, i.e. do not draw any card if a 2 or a 3 was played before;

- Action:
  - Queen cards: reverse the playing order from counter-clockwise to clockwise or clockwise to counter-clockwise;
  - Jack cards: skip the next player;
- Regular: any other card has no special effect and is only used to match the previous card's rank or suit;

Notes on cards and attacks:

- The effect of the attack cards is cumulative.
- A Jack, or Queen, of same suit as the previous card can be played to redirect an attack on another player;
- A player not playing any special card (2, 3, 7, Q, J) must draw cards from the stock pile, and this ends his turn;

For instance last week when the three sisters took a break during their revision fo the MATLAB exam they played *One Card* and the following scenario occurred. As Chiaki played "2 Diamonds", Kana "3 Diamonds", and Haruka "3 Spades", everybody expected to see Chiaki drawing eight cards, but she played a Queen such that in the end Haruka had to draw them.

2 Project goals and program structure

As Haruka, Kana, and Chiaki all agree on the importance of a good code structure they decide to follow the advice of their VG101 instructor to never start coding before precisely knowing what to do. Hakura in particular highlights that if they organise their code well they can easily adjust it in the future to add new features without rewriting much. Kana adds that during the C part of their course they will probably learn new things that could be helpful to their project. However it might lead to substantial rewriting of the code as their understanding of programming improves. Chiaki concludes that it is like in real life for developers: they write a program, then check what parts can be improved or made more efficient, then they adjust and fully rewrite some of the code until everything is as good as it can be.

To ensure they go in the right direction they start discussing what they want to achieve.

Project goals

The three sisters starts by thinking of the big picture, what they want there program to do. Obviously they need players and cards, so it makes sense to define options specifying the number of players, how many cards each one of them gets, and the number of decks to be used. Kana recalls that the number of rounds should also be flexible, so an option is also needed for that. Beyond those basic arguments they would also like their program to define a log file saving all the details of a game, and to feature a demo mode. Of course they do not forget the usual help explaining how to run the program and use the options.

```
sh $  ./onecard -h
  -h|--help               print this help message
  --log filename          write the logs in filename (default: onecard.log)
  -n n|--player-number=n  n players, n must be larger than 2 (default: 4)
  -c c|--initial-cards=c  deal c cards per player, c must be at least 2 (default: 5)
  -d d|--decks=d          use d decks 52 cards each, d must be at least 2 (default: 2)
  -r r|--rounds=r         play r rounds, r must be at least 1 (default: 1)
  -a|--auto               run in demo mode
```

Now that the three sisters have clearly defined the main lines of their project, they start thinking of the best approach to minimize their work. For instance Haruka emphasizes that the number of players participating in the game should not impact the clarity and complexity of their code. Similarly, there is no point in writing a completely different program or set of functions for the demo mode: it should take advantage of the regular functions used by other players.

Keeping these ideas in mind they want to define some generic output that can be easily generated independently of the number of players or running mode. A first step would be to always display card following a same order, this will clearly facilitate the choice of the players and the reading of the log file. After some discussions they all agree on the following arbitrary order for display:

$$\textit{Spades} < \textit{Hearts} < \textit{Diamonds} < \textit{Clubs},$$

and if two cards suits are the same, then use

$$2 < 3 < \cdots < 10 < \textit{Jack} < \textit{Queen} < \textit{King} < \textit{Ace}.$$

Once this issue solved, the sisters think of the playing experience. If we have several players on different computers then it is no problem to ask each of them to take turns and play. However if they use the same computer there is a need to redraw the screen after each turn to ensure the next player does not see the cards of the current one. That is an especially important detail that they will need to consider. In fact based on what they learnt in the labs it is very likely that refreshing the screen is done differently from one operating system to another, so redrawing the screen will probably require a bit of research to ensure a full compatibility with all of them. Aside of that, a simple question and answer format should be enough to display the previously played card, the user's cards and allowing him to play.

After some discussion they decide to go with this basic approach and structure their code well such that they can replace this basic User Interface (UI) by a more advanced and fancier one if they have time. At this stage the only thing left for consideration is the format of the log file, which is similar to what the demo mode should be displaying, itself being based on the regular multi-player version.

Hence they come up with the following format, to which they add comments to clarify their thoughts. Of course their comments, everything after a #, will not appear in their program...

```
########################
#                      #
# Welcome to One Card! #
#                      #
########################

---- Initial setup ----
Number of rounds: 1
Number of decks: 2
Number of players: 2

Shuffling cards...
Shuffle result:                # shuffle result only displayed in log and demo mode
Spades 2, Spades 10, Hearts 2, Diamonds A, Clubs 3,
Spades 7, Diamonds 8, Diamonds 9, Clubs 6, Clubs 8,
...                            # more results skipped here

---- Game ----
Dealing cards...
# only display current user for a real game, server and demo mode show all players
Player 1: Spades 2, Spades 10, Hearts 2, Diamonds A, Clubs 3
Player 2: Spades 7, Diamonds 8, Diamonds 9, Clubs 6, Clubs 8

First card: Hearts A
Player 1 plays: Heart 2
Player 1 cards: Spades 2, Spades 10, Diamonds A, Clubs 3
# clear screen here for a real game and show the previously played card (Heart 2)
Player 2 draws: Spades Q, Clubs A
Player 2 cards: Spades 7, Spades Q, Diamonds 8, Diamonds 9, Clubs 6, Clubs 8, Clubs A
Player 1 plays: Spades 2
Player 1 cards: Spades 10, Diamonds A, Clubs 3
Player 2 plays: Spades 7
...                           # more details skipped here
Player 1 plays: Diamands A
Player 1 wins!

---- Stats ----
Round 1 result:
Player 1: 0, total: 0
Player 2: -3, total: -3
Round 1 ends.
```

Once they precisely know what they expect to complete and how they could extend their game in the future, the three

sisters move on to the program structure. They design it to specifically fulfill all their goals.

Program structure

One of the fundamental concept when developing a project is *layer programming*. It brings much flexibility while also saving much rewriting when adjusting the code and allowing faster debugging in case of problem. The idea is to organise the code in term of layers and prevent any function for a lower layer to call functions from a higher layer. Functions from a higher layer can use functions in the same layer or a layer below.

In the case of their game the Haruka and Chiaki identify three main layers from lowest to highest:

Layer 1. Functions and structures that will needed for the well functioning of the game:

- Data structures definitions, e.g. cards, player;
- Basic function to handle the data structures;

Layer 2. Functions needed to play the game:

- General functions applying to all players (server):
  - Initialise a game;
  - Add players to a game;
  - Start, proceed, and end a game;
  - Prepare the game stats, e.g. scores;
- Function specific to each player (client):
  - A function to init a new player, with player information, e.g. score, cards, etc.;
  - A function to play a card according to the last card played by the previous player;

Layer 3. Function needed for the user to interact with the computer and play:

- A function to display the current status of the game, e.g. cards in hand, the previously played card, etc.
- A function allowing the player to choose what he wants to do;
- A Demo mode, showing a randomly generated game being played;

As Kana does not really understand why her sisters structured the program this way Haruka explains layer programming with an example. Keep in mind that the goal is to save time and render the program clearer. So now imagine that functions in Layer 1 could call functions in Layer 3. Then lets say a function from Layer 3 that used to take two `int` as input now takes one double and one `int`. Then any function that uses it must be rewritten, i.e. changes in Layer 3 might impact functions in Layer 1. This would be very bad since for instance changing the way a player interacts with the computer would mean redefining low level data structures which are also used in Layer 2. In other words the whole program would need to be rewritten, just for a simple adjustment!

Besides, since the third layer is the UI having it as the top layer, separated from the rest of the program, allows the programmer to write different kinds of UI. For instance a programmer could start with an text UI but then decide to implement a Graphical User Interface (GUI). With layer programming the UI has no impact on the rest of the program, so adding new types is very simple. Similarly once all the lower level functions are ready a demo mode is simple the exact same functions as for real users, but called with random parameters...

As emphasized by Haruka and Chiaki, their layers are to be used as rough guidelines and some minor adjustments might be needed, and new functions or data structures added. It is even possible to add more layers as long as no lower layer needs a higher one.

Now that kana fully understands the layers defined by her sisters they all start discussing what data structures to use. Unfortunately at the moment they do not know much so they decide to use simple arrays as in MATLAB. They have a very quick look at chapter 7 in their course and notice that arrays are easy to define and use, so it should not be a problem. However they feel something better can be done.

While arrays are good it seems a new concept is also introduced in chapter 7: *pointers*. As this seems more complicated it is worth starting with arrays and when they master pointers, after completing homework 6, change their code to use this new tool. Chiaki recalls what she mentioned earlier: it is common when programming to write some code and change it later to improve it, this is exactly what they should do when they learn about pointers.

After a bit of thinking the come to the conclusion that they would need two main types of data structure: (i) one that allows to quickly move from one player to another in a "circle", and can be reverted at no cost when a Queen is played; and (ii) one that can automatically resize itself to store all the cards in the discard and stock piles.

From what they understand in their programming course memory should not be wasted. In that context what is the point of having an array with 208 cards to accommodate four decks when the discard pile has only a few items? It would be perfect to automatically resize as it grows...

They heard from previous students that those ideas will be studied in homework 6, so they eagerly wait for its release, but in the meantime they use arrays to ensure they do not run into any last minute issue with their design and get more practice in C.

3 Project tasks and milestones

The project features three milestones, the last one corresponding to the final submission. Each milestone should take about a week to complete. For each milestone students must submit their current code with all the usual relevant files attached as well as with a short `Changelog.txt` file that describes the progress done since the last submission.

Milestone 1

Tasks to be completed:

- Define the card structure;
- Define the player structure;
- Accept command line arguments;
- Write a function to play a card;
- Write a function to draw a card;

Milestone 2

Tasks to be completed:

- Define the deck data structure using an array;
- Arrange all the players in an array;
- Write a function to shuffle decks of cards;
- Write a function to deal the cards to the players;
- Write functions for several players to play together;

Milestone 3

Tasks to be completed:

- Use a circular double linked list for the players;
- Use a dynamic array to handle the stock and discard piles;
- Complete the demo mode;
- Proof-read you code and ensure it is fully complying with the C standard;

Bonuses

Optional tasks bringing a reward:

- All players to play more than one card on their turn;
- Draw the cards using ASCII art;
- Use an external library to to draw in the terminal, e.g. ncurses;
- Use a toolkit to implement a GUI, e.g. GTK;
- Create a real game setup that can be played over a network;

4 Project submission

Before submitting the project on Canvas, ensure the project compiles on JOJ.

- A project that has not been submitted to JOJ will not be graded;
- JOJ will compile the code using the flags `-O2 -Werror -pedantic -Wall -std=c11`;
- A project that is not compiling on JOJ will not be graded;
- No test case is offered for the project, the goal of JOJ is only to ensure the written code compiles and complies with the C standard;
- If the Canvas submission includes a GUI, please contact the teaching team when uploading the code since it will not compile on JOJ;

5 FAQ

This section lists Frequently Asked Questions (FAQ).

1. I have no idea where to start and what to do.

   Log on Piazza and discuss with other students and the teaching team. Clearly explain what you do not understand, and why you feel stuck, **do not ask for a solution**. If several opinions appear to be valid determine which ones is the best and most reasonable. Document your choices in the README file. Feel free to edit or refine others' questions and answers. To ensure everybody benefits from the question and its answer **no question will be answered if not asked on the project discussion**.

2. I am very busy with the project and do not have time to work on the assignments.

   Change your work strategy: **first solve the assignments and then move on to the project**. Several exercises from the assignments can be partially reused in the project. Directly starting with a hard task is a waste of time. Assignments are designed to help you progress, and milestones have been organised with the assignments in mind.

3. Is there any easy and clean way to parse command line arguments? Look at the file `get_opt.h`.
4. I am expected to get my program to run on all the common Operating Systems (OS), but I am running only one of them. How can I check?

   In the basic version of the game, the only feature that require to call of an OS specific function is the redrawing of the screen. Search how to do it for each of the most common OS (Windows, Mac OSx, and Linux), then use the "family" of `#ifdef`, `#define` instructions to detect the running environment and ensure the correct function is used.
5. How should I provide the location of the logfile?

   A file location can be expressed using either a relative or an absolute path. As the absolute path is "computer specific" it is not a good idea to defined any absolute path in the program. Therefore in this project only relative path should be used inside the program. A user should however be able to use either an absolute or relative path when providing a file location as a command line argument.

A Develop a REAL game!

In real life, game server and clients are always separated in different programs, and distributed on different computers. They are connected by some protocols (eg. socket). If you've finished all of the tasks above, we'll give you a chance to implement a real game!

You can choose any way to implement separated server and clients, as long as they can communicate with each other. The minimum requirement to gain the bonus is that you can run a server and several clients in different processes. It's better (more bonus?) if your server and clients can be distributed on different computers.

The following guides are only for your reference, you don't need to follow everything in it. However, if you hope that your server and clients can communicate with those written by other one, both of you should carefully follow the API part below with socket communication.

Socket Communication

In practice, socket usually refers to a socket in an Internet Protocol (IP) network (where a socket may be called an Internet socket), in particular for the Transmission Control Protocol (TCP), which is a protocol for one-to-one connections. In this context, sockets are assumed to be associated with a specific socket address, namely the IP address and a port number for the local node, and there is a corresponding socket address at the foreign node (other node), which itself has an associated socket, used by the foreign process. Associating a socket with a socket address is called binding.

Once an address was bound to a socket, the program that created the socket starts to receive (listen) messages sent to the address. In the sample code we provided `socket.c, socket.h, and socket_demo.c`, you can learn more about how messages are sent and received.

API (Application Programming Interface)

In order to communicate between server and clients, they should both agree on a series of criterion, it is often called "API" (Application Programming Interface).

We will release the details later in this part.