Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Programming Language I • DIM0120

◁ Keno Game Programming Assignment ▷

29 de junho de 2021

1 Introduction

In this programming assignment you are going to write a program that plays a simple text-based version of the game Keno.

Keno is a popular gambling game with similarities to a lottery or bingo. Players place a bet and choose anywhere from one to twenty numbers between 1 and 80, inclusive. Once the players have chosen their numbers, the game generates twenty random numbers between 1 and 80, and players receive a payoff based on how many numbers they picked that matched the chosen numbers.

For example, if a player picked seven numbers and all seven were chosen (very unlikely!), she might win around $\$10.000$ for a ten dollar bet. The actual payoffs are based on the probabilities of hitting k numbers out of n chosen.

This assignment will introduce you to classes in C++ by challenging you to design and code a real world game application.

2 Game Overview

To begin a game, your program should be able to read a player's bet from an ASCII file. The player's bet file name should be passed into the program as arguments of the function `main()`, from the command line. Remember that the integer parameter `int argc` (argument count) is the number of arguments passed into the program, whereas the `char * argv[]` is a list with all arguments passed to the program by the user, including the program's name, stored at `argv[0]`.

Here is how you should provide a bet file to the keno program.

```
$./keno bet_12spots.dat
```

You find below an example of a valid bet file.

```
1500.0
3
21 12 64
```

A typical bet file has a set of three data lines, representing:

- The player's initial credit (`IC`), expressed as a real value.
- The number of rounds (`NR`) the game should run, represented by an integer.
- A set of up to 15 unique numbers in any order, separated by at least one white-space.

Your program should **validate** the bet file by ensuring that all the number the player picked, called *spots*, are in the proper range $[1, 80]$, and that each number appears only once in the bet file. This means that any repeated occurrences of a number in the *spot* list should be ignored.

The input file may have blank lines between valid lines, and leading white-spaces at the beginning of any valid line. In case the bet file contains more than 15 numbers, your program should consider only the first 15 *unique*[1] valid numbers, print out a warning message to the client explaining the problem found, and accept the bet as valid (assuming, of course, that at least one number is valid).

2.1 Gameplay

After a valid bet is processed, the game runs `NR` rounds, automatically waging in each round `IC/NR` credits. For each individual round the game should randomly pick 20 winning numbers, the *draw*, and display them on the screen. The player's bet numbers, which are called *spots*, are then compared to the *draw* numbers to determine how many of them match. The set of correct numbers picked by the player are called *hits*. Here is a summary of the these Keno-related terms:

- *spots*: the set of $1 \leq m \leq 15$ numbers chosen by the player in a bet.
- *draw*: the set of 20 numbers the game randomly generates; these numbers are unique, which means no repetition.
- *hits*: the set *spots* $\cap$ *draw*, which may have size $0 < n \leq 15$.

Assuming the game draws numbers fairly, the probability law states that the greater the size of the spots, the lesser is the chances of a player guessing right all the numbers. That is why Keno tends to pay off more to players who hits more numbers in a bet. In practical terms, the number of *hits* determines the *payout factor*, which, in turn, is multiplied by the round's wage to determine whether the player wins or looses credits.

The payout factor comes from the *payout table*. A payout table is chosen based on the number of *spots* present in a bet, and it determines the player's expected return, based on the number of *hits* obtained in a round. In short, we have

$$\textit{total payout} = \text{wage} \times \text{payout factor}.$$

Table 1 shows all the 15 possible payout tables.

[1] Does the `unique()` from the previous assignment ring a bell?

	Hits															
# spots bet	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	3														
2	0	1	9													
3	0	1	2	16												
4	0	0.5	2	6	12											
5	0	0.5	1	3	15	50										
6	0	0.5	1	2	3	30	75									
7	0	0.5	0.5	1	6	12	36	100								
8	0	0.5	0.5	1	3	6	19	90	720							
9	0	0.5	0.5	1	2	4	8	20	80	1200						
10	0	0	0.5	1	2	3	5	10	30	600	1800					
11	0	0	0.5	1	1	2	6	15	25	180	1000	3000				
12	0	0	0	0.5	1	2	4	24	72	250	500	2000	4000			
13	0	0	0	0.5	0.5	3	4	5	20	80	240	500	3000	6000		
14	0	0	0	0.5	0.5	2	3	5	12	50	150	500	1000	2000	7500	
15	0	0	0	0.5	0.5	1	2	5	15	50	150	300	600	1200	2500	10000

Tabela 1: The payout table for up to 15 *spots*. Each line of the table corresponds to a payout scale based on the number of *spots* in a bet (left most column). For example, suppose a player waged 100 credits on a 5 *spots* ticket and got 3 hits (for which the *payout rate* is 3); in this case the player would receive $100 \times 3 = 300$ credits back.

2.2 Text-based Interface

After a valid bet is read and processed, the game should display all the bet information on the screen, as well as the corresponding payout table. The spots **must** be presented in ascending order.

Next, the program should display, for each `NR` round, the 20 numbers randomly picked, the player's *hits*, the payout rate received, and the amount of credit won or lost. The draw **must** be presented in ascending order.

Finally, the program displays on the screen a summary of the game, showing the total amount of money the player won/lost after finishing all drawing rounds.

See bellow one possible text-based output for the bet presented in Section 2.

```
>>> Preparing to read bet file [data/bet_03.dat], please wait...
-------------------------------------------------
>>> Bet successfully read!
    You are going to wage a total of $1500 dollars.
    Going for a total of 3 rounds, waging $500 per round.

    Your bet has 3 numbers. They are: [ 12 21 64 ]
         -------+---------
         Hits   | Payout
         -------+---------
         0      | 0
         1      | 1
         2      | 2
         3      | 16
         -------------------------------------------------
         This is round #1 of 3, and your wage is $500. Good luck!
         The draw numbers are: [ 3 6 12 20 21 23 26 27 28 31 32 35 45 48 55 59 63 64 69 74 ]

         You hit the following number(s) [ 12 21 64 ], a total of 3 hits out of 3.
         Payout rate is 16, thus you came out with: $8000
         Your net balance so far is: $9000 dollars.
```

```
        ------------------------------------------------
        This is round #2 of 3, and your wage is $500. Good luck!
        The draw numbers are: [ 2 3 7 10 15 17 18 21 23 28 29 33 37 40 41 43 50 71 72 79 ]

        You hit the following number(s) [ 21 ], a total of 1 hit out of 3.
        Payout rate is 1, thus you came out with: $500
        Your net balance so far is: $9000 dollars.
        ------------------------------------------------
        This is round #3 of 3, and your wage is $500. Good luck!
        The draw numbers are: [ 4 8 10 16 20 23 28 30 32 34 45 46 50 51 52 63 64 68 74 80 ]

        You hit the following number(s) [ 64 ], a total of 1 hit out of 3.
        Payout rate is 1, thus you came out with: $500
        Your net balance so far is: $9000 dollars.
>>> End of rounds!
------------------------------------------------

===== SUMMARY =====
>>> You spent in this game a total of $1500 dollars.
>>> Hooray, you won $7500 dollars.  See you next time! ;-)
>>> You are leaving the Keno table with $9000 dollars.
```

3 Design and Implementation

You program should present at least one C++ class. Because we are beginning to experiment with basic Object Oriented Design principles, you should try to identify the *entities* that better represent or model the problem you are trying to solve. For each entity you need to define what are their role in the system, or how they interact whit one another (*class methods*), and what are their properties (*class attributes*).

For instance, you may identify a **Game Manager** entity that is in charge of

1. setting up the whole game before the simulation starts;
2. creating individual bets, one for each round;
3. managing each round, i.e. drawing the 20 numbers and checking them against the current bet;
4. calculating the pay out factor based on the hits the player guessed right, and;
5. keeping track of all the money the user won or lost while playing all the rounds.

At a lower level, another entity is the **Keno bet**, that emulates a single bet that is placed each round. The actions of a bet are

1. receiving and storing spots from the game manager, making sure to accept only valid number with no duplicates;
2. receiving and storing the wage for that bet, and;
3. checking out which numbers are right, given a list of drawn numbers, and returning those numbers as a list of hits.

We suggest to model a Keno bet as the `KenoBet` class, which may follow the organization suggested in Code 1.

Code 1 The `KenoBet` class header.

```
using number_type = unsigned short int; //<! data type for a keno hit.
using cash_type = float; //<! Defines the wage type in this application.
using set_of_numbers_type = std::vector< number_type >;

class KenoBet {
    public:
        //! Creates an empty Keno bet.
        KenoBet( ) : m_wage(0)
        { /* empty */ };

        /*! Adds a number to the spots only if the number is not already there.
            @param spot_ The number we wish to include in the bet.
            @return T if number chosen is successfully inserted; F otherwise. */
        bool add_number( number_type spot );

        /*! Sets the amount of money the player is betting.
            @param wage_ The wage.
            @return True if we have a wage above zero; false otherwise. */
        bool set_wage( cash_type wage );

        //! Resets a bet to an empty state.
        void reset( void );

        /*! Retrieves the player's wage on this bet.
            @return The wage value. */
        cash_type get_wage( void ) const;

        /*! Returns to the current number of spots in the player's bet.
            @return Number of spots present in the bet. */
        size_t size( void ) const;

        /*! Determine how many spots match the draw set passed as argument.
            @param hits_ List of hits randomly chosen by the computer.
            @return The list of hits. */
        set_of_numbers_type
        get_hits( const set_of_numbers_type & draw ) const;

        /*! Return the spots the player has picked so far.
            @return The player's spots picked so far. */
        set_of_numbers_type get_spots( void ) const;

    private:
        set_of_numbers_type m_spots;  //<! The player's bet.
        cash_type m_wage;             //<! The player's wage
};
```

While developing the game, try to use as much as possible the functions from the `<algorithm>` library. They are design to simplify typical programming tasks (called *boilerplate code*) in an efficient fashion.

Make sure to extensively test your program. Run several test cases, with valid and invalid bet files. You may also ask friends or colleagues to test out your program and give you some valuable feedback regarding the user interface, and the fun factor.

4 Project Evaluation

Your task in this assignment is to design and develop a complete program that implements the Keno game as described in this specification document. Any doubts related to parts of the game description that is not clear or any game-related information that might have not been well understood must be cleared with the instructor. Your program should not have any compiling errors, must be tested and fully documented. The assignment will be credit according to the following criteria:-

1. Receives input data via command line (5 credits);
2. Correctly handles the input bet file, treating both regular and problematic cases accordingly (20 credits);
3. Codes correctly the `KenoBet` class, according to the description provided in Code 1 (25 credits);
4. Executes correctly the amount of rounds defined in the input bet file (15 credits);
5. Identifies correctly the *hits* and the player's payoff for every round (20 credits), and;
6. Displays correctly the information requested in Section 2.2 (15 credits).

The following situations may *take credits out* of your assignment, if they happen during the evaluation process:-

- Compiling and/or run time errors (up to -20 credits)
- Missing code documentation in Doxygen style (up to -10 credits)
- Memory leak (up to -10 credits)
- Missing README file (up to -20 credits).

The README file (Markdown file format recommended here) should contain a brief description of the game, and how to run it. It also should describe possible errors, limitations, or issues found. Do not forget to include the author(s) name(s)!

Good Programming Practices

During the development process of your assignment, it is strongly recommend to use the following tools:-

- Doxygen: professional code documentation;
- Git: version control system;

- Valgrind: tracks memory leaks, among other features;
- gdb: debugging tool, and;
- Makefile: helps building and managing your programming projects.

Try to organize you code in several folders, such as `src` (for `.cpp` files), `include` (for header files `.h`), `bin` (for `.o` and executable files) and `data` (for storing input files).

5 Authorship and Collaboration Policy

You may work in a pair or alone. If you work as a pair, comment both members' names atop every code file, and try to balance evenly the workload. Only one of you should submit the program or the project's git repository via Sigaa.

Any team may be called for an interview. The purpose of the interview is twofold: to confirm the authorship of the assignment and to identify the workload assign to each member. During the interview, any team member should be capable of explaining any piece of code, even if he or she has not written that particular piece of code. After the interview, the assignment's credits may be re-distributed to better reflect the true contribution of each team member.

The cooperation among students is strongly encouraged. It is accepted the open discussion of ideas or development strategies. Notice, however, that this type of interaction should not be understood as a free permission to copy and use somebody else's code. This is may be interpreted as plagiarism.

Any two (or more) programs deemed as plagiarism will automatically receive **zero** credits, regardless of the real authorship of the programs involved in the case. If your project uses a (small) piece of code from someone else's, please provide proper acknowledgment in the README file.

6 Work Submission

You may submit your work in two possible ways: via GitHub Classroom (GHC), or, via Sigaa submission task. In case you decide to send your work via GHC you also need to send a text file via Sigaa submission task with the github link to your repository.

I any of these two ways, remember to remove all the executable files from your project before handing in your work.

◀ The End ▶