LETS MAKE A DEAL
GAME SHOW
SIMULATION

By Andrew Yaros
CS171-069 - Winter 2017
Homework 4

USER MANUAL

This program is a simulation of the old game show, *Lets Make a Deal* with host Monty Hall. It simulates the game many times to determine the best strategy for winning it. To use the program, simply run the executable; all the rest is done automatically.

There are three doors in the game. Two will have goats behind them, and one will have a car. The object of the game is to pick the door with the car.

[G] [C] [G]

When you first guess a door, Monty will pick a second one – the door he picks will have a goat.

[G] [C] ~~[G]~~

You now have the option to keep your original guess, or to choose the third door which neither of you has chosen. The goal of this program is to determine which of those strategies is more likely to win you the car.

The program will simulate the game many different times and calculate how often each strategy wins. It will show you a table of percentages; these are the results of some of the calculations. At the end of the program, it will show the average of all the results in the table, how likely you will win if you try either strategy, and which strategy is more likely to win you a car.

SYSTEM MANUAL

Overview

This program simulates the game show *Let's Make A Deal* (with host Monty Hall). The format of the game is described in the user manual on the previous page. This function uses the following libraries:

- `iostream`
- `iomanip`
- `cstdlib`
- `vector`
- `string`
- `ctime`

There are several global variables:

- `char C` – the character "C" (for "car")
- `char G` – the character "G" (for "goat")
- `const double HOW_MANY_DOORS` – the number of doors in the game. It should always be set to 3.0
- `const double PERCENT_NUMBER` – this is used for calculating percentages and should always be 100.0

The program simulates the game in batches of 10000; this is termed an experiment. At the end of an experiment, the percentage of games won is calculated. During each experiment, the game is tested 5000 times for each of the two strategies (this value may be changed).

- The player keeps their first guess
- The player changes their guess to the remaining door

Many experiments are run; some of results are put into a table for the user. Lastly, the average percentages from all experiments are then calculated and displayed to the user.

Note: Doors are defined by two pieces of information:

- The specific number of the door; this can be 1, 2, or 3. Used when checking whether a specific door has already been chosen.
- The character of the door – this can be `G` or `C` (whether a goat or car is behind the door, respectively. Used for testing whether a specific door has a car behind it.

Separate variables are used for the number of a door and the character of a door. Please keep this information in mind as you review the following functions and procedures:

---

int `random_num`

This function generates a random integer between 1 and a specified number. The sole parameter is `numberOfDoors` – the number of doors in the game; this function should always be called with `HOW_MANY_DOORS`, which is always 3.

It is used by `setupDoors` for choosing which door will have a car behind it, and by `pickDoorChoices` and `guessAgain` a door is chosen.

---

void `setupDoors`

This procedure determines which doors have goats behind them and which door has a car behind it. Each door is assigned a character value of `C` or `G`.

Parameters:

- `char &door1` – the character value of door 1
- `char &door2` – the character value of door 2
- `char &door3` – the character value of door 3

The function begins by assigning all the doors to `G`. A variable, `randomDoor`, is created, and `random_num` is used to assign it either 1, 2, or 3. If/else statements examine `randomDoor` and determine which of the three doors will be assigned `C`.

---

void `assignDoorCharacter`

This function is used in `pickDoorChoices` and in `guessAgain`. It uses the following parameters:

- `char door1` – the current character variable for door 1
- `char door2` – the current character variable for door 2
- `char door3` – the current character variable for door 3
- `int doorVariable` – the integer variable of a door which was chosen by either Monty or the Player. **The goal of this procedure is to find the appropriate character for this door.** *We will choose from one of those three character variables I just mentioned and put it into the following reference parameter:*
- `char &charVariable` – …I guess this procedure could have been a function which returned `charVariable`, but I decided to write it this way instead for some reason

If/else statements check the value of `doorVariable` and assign either `door1`, `door2`, or `door3` to `charVariable`.

void `pickDoorChoices`

This function determines which door the player choose and which one Monty chooses. It uses the following parameters:

- `char door1` – the current character variable for door 1
- `char door2` – the current character variable for door 2
- `char door3` – the current character variable for door 3
- `int &doorPlayer` – reference parameter containing the integer value of the door the player picks
- `int &doorMonty` – reference parameter containing the integer value of the door Monty picks

First, the player chooses a door; `random_num` assigns a value to `doorPlayer`.

Right now, we don't need to know what was behind the player's door. We do need to know what is behind Monty's door; he's not allowed to pick one with a car behind it. Thus, a character variable for Monty's door (`montyDoorChar`) is created.

A while loop is used when Monty chooses a door: `doorMonty` is assigned a value by `random_num` and `montyDoorChar` is assigned a character by `assignDoorCharacter`. The loop continues while either of the following remains true:

- `doorPlayer` == `doorMonty` (Monty can't pick the door the player picked)
- `montyDoorChar` == `C` (Monty can't pick a door with a car behind it)

When the door Monty picks satisfies those conditions, the loop ends. The procedure returns the door choices through the reference parameters.

bool `didPlayerWin`

This function checks to see if the door the player picked won. It uses the following parameters:

- `char door1` – the current character variable for door 1
- `char door2` – the current character variable for door 2
- `char door3` – the current character variable for door 3
- `int doorPlayer` – the value of the door the player picked

A character variable (`doorCharacter`) for the player's door is created. An if statement checks to see if the character is `C`. If so, the player won, and the function returns true. Otherwise the function returns false.

int `guessAgain`

This function picks the remaining door (which hasn't been chosen by the player or Monty). This remaining door is the player's second guess. It returns the integer value of the remaining door.

- `char door1` – the current character variable for door 1
- `char door2` – the current character variable for door 2
- `char door3` – the current character variable for door 3
- `char doorPlayer` – *the character value* of the door the player picked
- `char doorMonty` – *the character value* of the door Monty picked

An integer `secondGuess` is declared – this will be the value of the remaining door. Within a do while loop, `secondGuess` is assigned a number by `random_num`. The loop continues until `secondGuess` is not the same as Monty's guess *or* the player's guess.

The function then returns the value of the remaining door.

double `guessSimulation`

This function simulates the game a specified number of times and returns how many times the player wins. The parameters are:

- `int numberOfGames` – the number of times the simulation runs
- `bool secondG` – whether the player should make a second guess each time

First, variables are set up:

- `door1`, `door2`, and `door3`: character values for each of the doors
- `playerDoor`, `montysDoor` – integer values for the doors chosen by the player and by Monty
- `gameResult` – for use with `didPlayerWin`
- `secondGuess` – used to hold the integer value of the second guess, if applicable
- `gamesWon` – counter variable, set to 0

Next, a for loop simulates the games `numberOfGames` times. Each cycle of the loop is a new game.

- `setupDoors` assigns characters to each of the doors
- `pickDoorChoices` picks doors for the player and Monty
- If `secondG` is true, the player makes a second guess; `guessAgain` returns the remaining door to `secondGuess`
- `didPlayerWin` checks whether the player won. If yes, then one is added to `gamesWon`.

After the loop, a double, `percentageWon`, is created. The percentage of games won is calculated: (`gamesWon`/`numberOfGames`) * 100

The percentage is returned, and the function ends.

Main program

The program begins by specifying the seed for random number generation (this is based on the current time and does not currently meet US Department of Defense standards).

Explanatory text is printed out for the user.

A constant is defined for the number of experiments to be run (`HOW_MANY_EXPERIMENTS`). This is set to 100, *but may be changed by the programmer.*

A constant is defined for the number of games to be played during each experiment (`HOW_MANY_GAMES`). This is set to 5000 (half of the number we want, 10000 – we need half the experiments to be for the first guess scenario and half for the second guess scenario). *This constant may also be changed by the programmer.*

Doubles `firstPercentage` and `secondPercentage` are created to hold the percentage results of each scenario. Doubles are also created to calculate the average of all percentages (`firstTotalPercent`, `secondTotalPercent`). Vectors are created to hold percentage values for each type of scenario (`firstGuessVector` and `secondGuessVector`).

We are using all the experiments to calculate the average percentage values. However, we are going to draw a table for the user and *do not intend to use all the results in this table*.

`tableLengthFactor` determines the length of our table – it is set to 4 by default but can be changed. The number of rows in the table is 100 / `tableLengthFactor`.

`resultRetentionFactor` will be used in our main experiment loop when adding results to the table. It is based on `tableLengthFactor`. At the default values, it may appear redundant, but it is necessary if one increases or decreases the number of experiments run. It is calculated by dividing (`tableLengthFactor` * `HOW_MANY_EXPERIMENTS`) by 100.

A counter, `vectorCounter`, is created to add items from our experiment into to the vectors.

We are now ready to run experiments. This is done with a for loop which runs `HOW_MANY_EXPERIMENTS` times.

Within this loop, `guessSimulation` is used to get percentage values for each scenario. Next, we add these results to the table if n is a multiple of `resultRetentionFactor`. Finally, the percentage values are added to `firstTotalPercent` and `secondTotalPercent`. The loop then repeats.

After the loop ends, `firstTotalPercent` and `secondTotalPercent` are used to get the average of all percentages calculated during the experiment. Those values are stored as doubles (`firstAveragePercent`, `secondAveragePercent`) and printed out for the user later.

Next, the table is printed for the user. Column widths are set, more information is printed to the user, and a for loop draws each row of the table using the values in `firstGuessVector` and `secondGuessVector`.

Finally, the average percentages are shown to the user. An if statement compares the two and tells the player which strategy is more likely to win. (This is always going to be the second strategy, unless the laws of probability somehow magically change).