Youssef Rizk

March 9, 2015

**Motivation for Program Operation and Structure**

The program written to produce the famous “2048” game has adopted a very systematic and methodical approach, whereby the same general functions were defined for each of the four possible movements. The motivation behind such a means of writing the program comes from the simplicity of making amendments to the code if need be, as well as much easier debugging. Also, the variations of the functions for different directions of movement are rather minute, which allows for easier coding. These general functions will be outlined in what follows. Two vectors of type “int” are used to store the entries that make up the game configuration. One such vectors is “original”, which contains the game set-up of the configuration before any movement is applied and is used as the input to most initial functions. The vector “Modified” contains the game set-up after the movement has occurred, and is modified each time it is called by reference as an input to a function.

A first function was defined that takes in input a row and column, and produces an integer whose value corresponds to the index of the entry at that certain row or column in the vector that stores all entries. It is important to note here that counting rows and columns in this program starts from 1 rather than 0, simply for convenience because we are more used to counting from 1 rather than 0. Thus, rows and columns are numbered 1-4.

The first set of functions is designed to modify the vector containing the pre-movement configuration to reach the final setup, ultimately returning the modified vector. A general Boolean function, “zero\_in\_between”, has been created to check whether there are two numbers in the same row/column separated only by 0’s (representative of blank cells). The function “check\_if\_same\_entries(direction)” then calls on “zero\_in\_between” as it determines whether there are two numbers in the same row/column that are separated by 0’s. If “zero\_in\_between” returns true, then “check\_if\_same\_entries(direction)” places these two numbers next to each other in the same row/column. As this function involves modifications to vectors, they are of type “void” and have the modified vectors passed by reference as output parameters.

The next function, “add”, simply takes the “Modified” vector (after calling on “check\_if\_same\_entries(direction)”) and adds the entries in the columns for vertical motion, or those in the rows for horizontal motion, if the entries are the same number. This is done by making one of the entries the sum of the two, while setting the other equal to 0. This, of course, does not happen randomly, but depends on the exact movement command.

Another major function, “move(direction)”, handles the exact motion of the entries. It is achieved by employing a “for” loop that places non-zero entries adjacent to each other along the row/column. Of course, once an entry is moved to be adjacent to another entry, its original location is replaced with a 0.

Finally, the general function “command(direction)” employs all of the previously structured functions. Firstly, it sets the modified vector equal to the original one, such that the pre-movement game configuration is stored in “Modified” and can then be altered. It calls the function “add” for all the rows/columns of the game setup, which results in the modified vector now containing the new configuration with all possible additions. With the additions already made, this function calls “move(direction)”, which handles necessary movement of the entries to achieve the almost final configuration.

The final group of functions focuses on establishing whether a given movement will be at all possible. The functions “no\_zero” and “no\_zero\_horizontal” are employed to determine whether there are any zeros in a given row/column or not. The general function “not\_same” checks whether two adjacent entries in a column (or row in the case of “not\_same\_horizontal”) are the same, for if they are, movement is possible. A final function, “last\_not\_zero\_down”, checks, if there are zeros in a column, whether or not they are all aligned at the top (or similarly for the different directions). The functions “first\_not\_zero\_up”, “left\_not\_zero\_right”, “right\_not\_zero\_left” accomplish a similar task for different movements. At this point, the only possibilities for which a certain movement will not be possible is if all the entries in the row/column are non-zero and no adjacent terms are similar, or if all entries are non-zero and all zeros are concentrated at the end of the row/column opposite to the direction of movement. All these functions return a Boolean variable as output. A general Boolean function, “check\_if\_space(direction)”, then takes in input the vector containing the game setup and checks whether a certain movement is possible or not by combining the three previously mentioned functions. The motivation for employing such a structure in the program stems from the easily broken-down and systematic way of checking whether a certain movement is possible or not.

A function called “check\_possiblilty” is used to determine whether movement in any direction is possible or not, for if not, the game is over. It does this by employing the “check\_if\_space” function unique to each direction of motion. If all these functions return false for all rows and columns, the game terminates.

The actual functions employed for each of the four possible movement directions follow an identical approach.

A random 2 is added using a function called “random”, which employs the “rand()” function in the <cstdlib> to produce two random numbers between 1 and 4, one number representing a row, and the other a column. If the entry in the location specified by the random row/column combination contains a 0, then it is replaced by 2 and the function is terminated. Otherwise, the function recursively calls itself.