2048 Game
Description and Motivation of Design Choices

<u>Design Specification</u>

- Ask for an input file from the user, if the file is found it will use this file as a starting position in the game. If the file isn't found, the program will use the default starting position.
- Input is given in the command line by typing (w, a, s, d) to move the grid in a specific direction.
- After the grid is moved, a 2 will randomly be placed on to the grid at a free location.
- If a move cannot be made, a 2 will not randomly be placed and the grid will not be printed.
- The game should only terminate once there are no moves available anymore.

<u>Design Choices</u>

The main purpose of the design choices in this project were made so that the game is as flexible as possible and that the different algorithms in the game work for a more general case and are not constrained by the 4x4 grid and base 2. Although this creates slightly longer code, it makes this code much more understandable as it uses general variables and therefore one can't assume that the size of the grid is a specific variable.

To implement this I have used the `#define a b` directive which makes the compiler replace all the instances of `a` by `b` in the code. Using this directive I have replaced the `char: w, a, s, d` by their corresponding directions and also replaced the integer defining the grid size by `GRIDSIZE` and the integer defining the base by `BASE`.

Another design choice that was made is the separation of the code in multiple functions that would each perform one specific operation in the program, such as moving the tiles, or seeing in what direction the tiles should be moved. However, all the printing to the screen is done in the main function apart from printing the grid, where it was simpler to create a separate function that prints out the whole grid.

The movement of the tiles should also be handled by only one function and should be as simple as possible without needing too many different conditional statements and just being able to move the tiles based on the arguments provided to the function.

Main Function

The purpose of the main function is to print out information onto the command line and initialise the parameters used in the program. It also runs the main loop of the game. The Main function also sets the seed for the random number generator to place the 2.

Movement of the Grid

I use two functions to move the grid around and tried to optimise these functions as much as possible so that I only had to really use one function to perform the movement and could use the other function to set up the movement so that it would go in the right direction. I first call the `moveGrid` function which looks at the character that the user entered and finds out in what direction the movement has to go. Depending on the direction it will then create a custom array, which is declared through a pointer and a specific separation. This enables the use of only one function to do the merging of that array as we can use the pointers that are separated by that specific amount to manipulate the data and the grid. Figure 1 demonstrates how the `moveGrid` and the function works for a 3x3 grid.

2D Grid	Column 1	Column 2	Column 3
Row 1	0x7fff8a8f56b0	0x7fff8a8f56b4	0x7fff8a8f56b8
Row 2	0x7fff8a8f56bc	0x7fff8a8f56c0	0x7fff8a8f56c4
Row 3	0x7fff8a8f56c8	0x7fff8a8f56cc	0x7fff8a8f56d0

As there is only one algorithm in the `mergeArray` function the two inputs of the separation and the start of the array have to be in the correct order. If the user wants to move the grid upwards, the inputs to the `mergeArray` function would be `0x7fff8a8f56b0` for the array and separation `3`. The separation is 3 and not 12 as the `sizeof(int)` will automatically be multiplied to the separation when manipulating pointers.

`0x7fff8a8f56b0` → `0x7fff8a8f56bc` →

When wanting to move the grid to the left the first pointer for Row 1 would be `0x7fff8a8f56b8`, and separation `-1`. The array for Row 1 would therefore be:

→ →

The merge function is called with the two arguments of the pointer and the separation. The merge function then loops through the array and copies all the non-zero integers from the custom array to a new array where each element is initially set to 0 and is also one larger than the grid size. This is because when we shift the values up after adding two together, the last value in the array has to be set to 0 as well. Then the algorithm checks if there are any two numbers next to each other that have the same value, and if they do it adds them together and shifts all the other values upward in the array. After it has gone through the whole array, it will just set the values to which the original custom array pointed to equal the values that are in the new array. As we are dealing with pointers it will automatically make the changes to the grid.

Adding the 2

I used one function to add the 2 to a random location on the grid and decided to use a 2D array to store the location of all the zero's in the grid so that it is then very easy generate a random number and place the 2 at the location where the zero was. It could largely be optimised by using pointers as in the movement of the grid, however this would have made the coding of it much harder and decreased the legibility.

Checking for change in grid

To satisfy the condition of not printing anything and not adding the random 2 when the grid does not change, I just copied the whole 2D array and then performed the user's movement on one of them and if they were different after the movement the algorithm then adds the random 2 and prints the grid. This could have also been more memory efficient by perhaps checking if the grid moves when actually performing the movement in the `mergeArray` function, however this doesn't improve the performance of the game enough to want to use that method.

Checking game end

To check when the game ends, we have to see if there are any zero's left or if there are any numbers that are side by side and equal in value. To make this function more efficient I first start by checking if there are any zero's left and if there are the function returns the result immediately. Then I check around each other tile to see if the values are equal and immediately return if they are. This function will therefore only be majorly inefficient at the very end of the game when it has to check all the different squares, but since it is at the end it doesn't affect the performance of the game.

Conclusion

This program reaches all the design goals set in the specifications and uses minimal amounts of code for the merging and moving of the grid. It is a bit memory inefficient in some places, for example duplicating the whole grid to see if there was any change, however, this is a compromise for legibility and understanding of the code. This game can also be played with any size of a grid or the base of the game by just adjusting the define directives at the start. This adds a lot of customisability to the program and flexibility of the code.