Christian Michel & Chris Benka
Easy as ABC:

Overview:
The Easy AS ABC program reads from a text file written by the user in a specified format given to the user through a print command executed by the program. The program reads the text a line at a time, first using the contents of the first line to generate a 36X36 board and store the number for which the user demands the program to output the correct letter after the board has been correctly solved. The program using the constructed 36X36 board, then constructs the 4x4 grid to be solved, solves the board, and then finally returns the correct letter at the specified location. Upon completing these steps for the first line written by the user in the text file, the program rewrites the remaining lines to the user's file, effectively deleting the line we used to find the letter at the specified location. The program continues to execute these steps until there are no more lines to be solved. We selected the Easy As ABC program, because we had both designed a program to solve a soduko board, and felt that our knowledge from doing so would aid us in our design of our Easy As ABC program.

Algorithm for Reading Board:
Each line read from the source text file corresponds to one board of the Easy as ABC game. The loadgrid method called in the main reads in the first line, which has the positions of the filled cells and letters, from "Text.txt" using fgets. Then the algorithm reads in the remaining boards from the text file and stores it in "rest.txt", which will be used to temporarily store the contents of "Text.txt". When fprintsf is called, the contents of "rest.txt" are stored in "Text.txt" allowing the function to be called for each board without losing its content. The function then converts the tokens read in from the line into long its and stores them in the "position" array, while adding any letters that appear into the "letters" array. Furthermore, the function checks that all inputs are "A","B","C", commas, or digits between 0 and 9 before storing them in the arrays.

Next, loadgrid begins constructing the 6x6 board by reading in the first 4 lines of the board, which correspond to the position of the X's and storing them in their correct position in the boards. Then goes into a for loop that adds the letters into their correct position given by the number next to the letter. The loop also checks for letters that already appear within the inner grid that are considered invalid.

When constructing the 4x4 grid, the algorithm first places 'X's found in the 6x6 board and places them in their corresponding position in the 4x4 grid. Then the algorithm traces through the outer layer of the 6x6 board until it finds a letter, which it will place in cell next to it into the outer layer of the 4xr grid. However, if that cell is already filled, the letter will simply skip over the filled cell.

Lastly, the algorithm uses a for loop to place the 4x4 grid into a 1d array "onegrid" that can be returned to the main. The final position is added to 97 and stored in the int output, which is the cell the user wants the letter of. The output is now capable of being read as char and is stored in

the onegrid[16]. The message then returns a pointer to onegrid, which contains the output cell and the entire grid.

Algorithm for Solving Board:
Once the board was created it was necessary to begin solving the board to allow us to investigate what Letter was at the specified location. We discovered that the board was best solved similar through finding a spot on the board where there exists only one possible solution, or one possible letter to be placed at the spot. This occurs when there are 2 letters already placed in a row and when there are 2 letters are already placed in a column. As we started filling the letters in we always wanted to check these two conditions. If none of these conditions proved true then it was necessary to find a blank space represented by ‘M’ and examine the letters that are in its row and the letters that are in its columns.

The Algorithm can be broken down into four steps, all of which are controlled by the method findSol().

1. Check to see if the specified location's letter has been filled in,if so set char final to the letter found at the specified location, which will be later returned. Only if the letter has not been filled in at the specified location continue to step 2.

2. Check Rows if there exists only one possible solution to replace an 'M'(case that there are 2 letters in the row). Upon the first case of this proving true, replace 'M' with appropriate letter and jump back to step 1. If there exists no such case continue to step 3.
3. Check Columns if there exists only one possible solution to replace an 'M'(case that there are 2 letters in the column). Upon the first case of this proving true replace 'M' with appropriate letter, and jump back up to step 1. If there exists no such case continue to step 4.
4. Find anyplace there exists an 'M', parse 'M' 's col and rows. Replace 'M' with the letter that was not found during the parsing of 'M"s rows and cols. Jump to Step 1.

Robustness
Before playing solving the game, the code ensures that the grid is created properly and adheres to the rules of the game. User input error is typically the cause of invalid boards, so we initially checked for characters other than “A”,”B”,”C”, and commas and the digits 0-9 that appeared in the text file, returning an error message if these are found. Furthermore, our algorithm checks if there are more letters present on the board than indicated in the input file, which is the 5th number in each line. Our loadgrid also returns errors if the positions of the ‘X’s, given by the first 4 numbers in the line, are on the outer columns and rows of the grids. The code also returns an invalid board error if the position of letters found in the text file are within the inner grid before the 4x4 is actually constructed. Lastly, the loadgrid algorithm checks that a valid character (A,B,

or C) is in the correct position in case the user accidentally placed a digit next to the position of the letter; for example, if the user input all numbers instead of letters. If the user input errors mentioned above are present in the “Text.txt” file, the code will goto wrongInput, which stores a NULL onegrid that’ll print out a message in the main.

Debugging/testing:

Our debugging consisted of stepping line by line and comparing the various values of the arrays we used with expected values. Through this process we were able to make the necessary changes necessary, particularly in regards to dereferencing when reading in the text, for the program to execute as expected. We struggled with reading the text line by line, so as to allow us to solve it line by line. Using a combination of online documentation and the class textbook , and through a series of trial and error we found it best to use reading and writing to the same file.

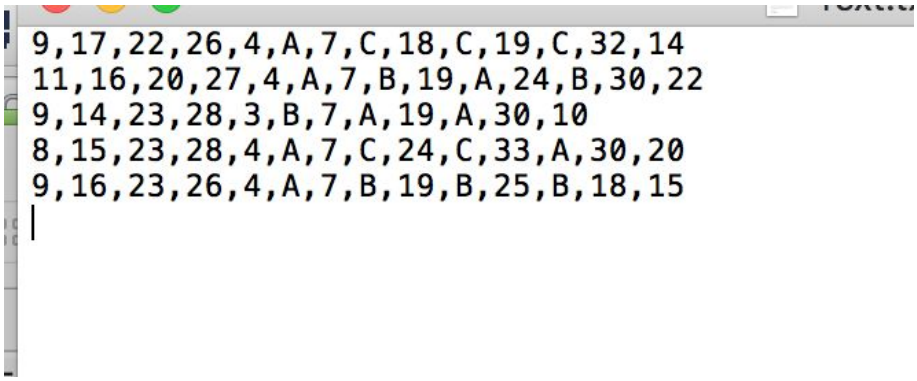
We encounter several issues with our pointers that returned segmentation faults during the compilation of our algorithm. Our first approach was to use xcode’s “activate console” feature to print out the pointers themselves, as well as what they were pointing at to ensure that the values we thought we were using were correct. Furthermore, we utilized breakpoints to determine what changes were made to the value of the pointers while running the code. We ultimately realized that a majority of our pointers and arrays, such as buffer and board, were not initialized to NULL, which was the root of our problem.

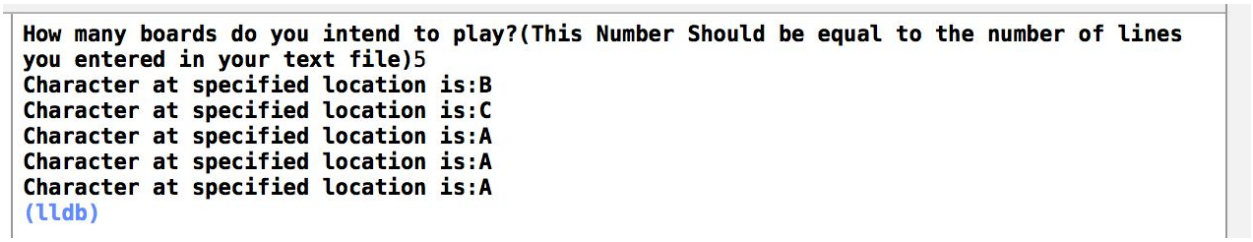
Another problem we encountered was determining the most efficient method of reading in the boards, line by line, without accidentally deleting them and keeping track of where the new lines were. We found that fscanf had a particular limitation that prevented us from interpreting new line characters properly and fgetsf would only read from the file until a new line character was found. Our solution was to utilize both fscanf and fgets to circumvent our issue with parsing the boards line by line, essentially using fgetsf purely for the first line and fscanf to read in the remaining lines. However, when it came time to stop reading in lines when all boards were read from the “Text.txt” file, fgets did not read in the NULL as we had intended. It instead read in a newline character that we couldn’t check for without interfering with the parsing of earlier boards in the code (if we had placed an if statement to break the code whenever a new line character was reached, the code would never read in a board). The solution to our problem was actually similar to that of the segmentation fault solution. We initialized the buffer to NULL and placed a condition for fgetsf to no longer read any lines when it is NULL, thereby allowing us to create a condition where buffer would not be updated and could, in fact, indicate whether there were any more boards to be read from the “Text.txt” file.

How to run it/how to compile it.

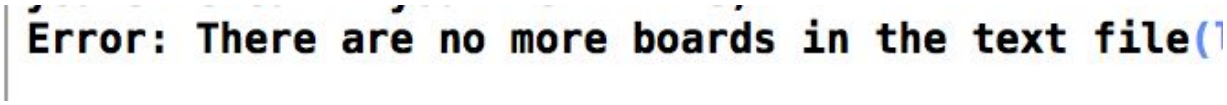
In order to compile our code:
1) Write the board positions in “Text.txt”, ensuring that only “A”,”B”,”C”, commas, and the digits 0-9 appear. It is also essential that the “Text.txt” files and “rest.txt” files used to construct the grids are in the same file/directory as the c files and headers.
2) Compile using the command gcc -Wall -std=c99 load.c solver.c main.c -o EASY
3) Run code using ./EASY

Proof that it works

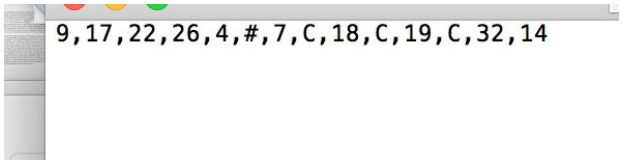




Nothing in text file:



Formatting Error:



```
How many boards do you intend to play?(This Number Should be equal to the number of lin
you entered in your text file)1

problem in board
```