Wordle

Presented by:

AKOUIRADJEMOU OUAIL ABDERRAOUF

Introduction

This is a wordle mini project that focuses on building a command line interface wordle game. it is mainly split into two parts. In the first part the user is given a chance to solve the game. However, if he fails, an automated solver will step in and try to solve it using minimum number of guesses possible.

The points that will be covered in this report:

1- The code structure

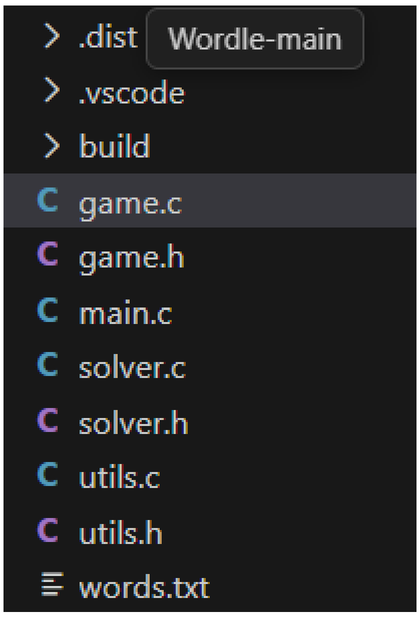
2- detailed explanation of the game logic

3- statistical comparaisons

4- detailed explanation of how the solver works

The code structure

The code is split into two type of files, C programs and header files. each header file contains references to the C files functions. this improves the code readability and reusability of the functions in general.



```
> .dist  Wordle-main
> .vscode
> build
C game.c
C game.h
C main.c
C solver.c
C solver.h
C utils.c
C utils.h
  words.txt
```

Here is a brief explanation of the purpose of each C file:

game.c: contains the base game and feedback logic.

solver.c: Contains the solver logic.

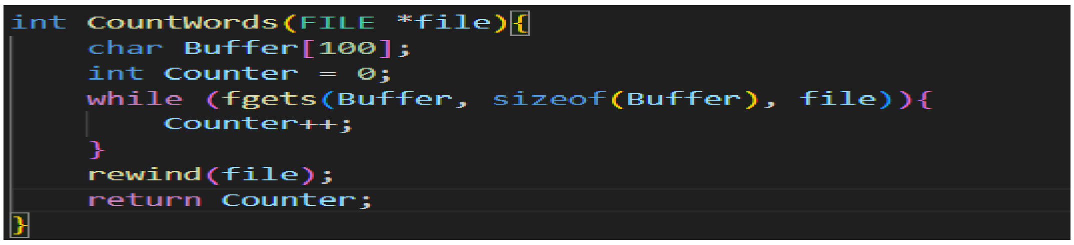
utils.c: Covers many functions that will be shared across both the game and solver files.

main.c: takes user input and starts the game or the solver

explanation of the game logic

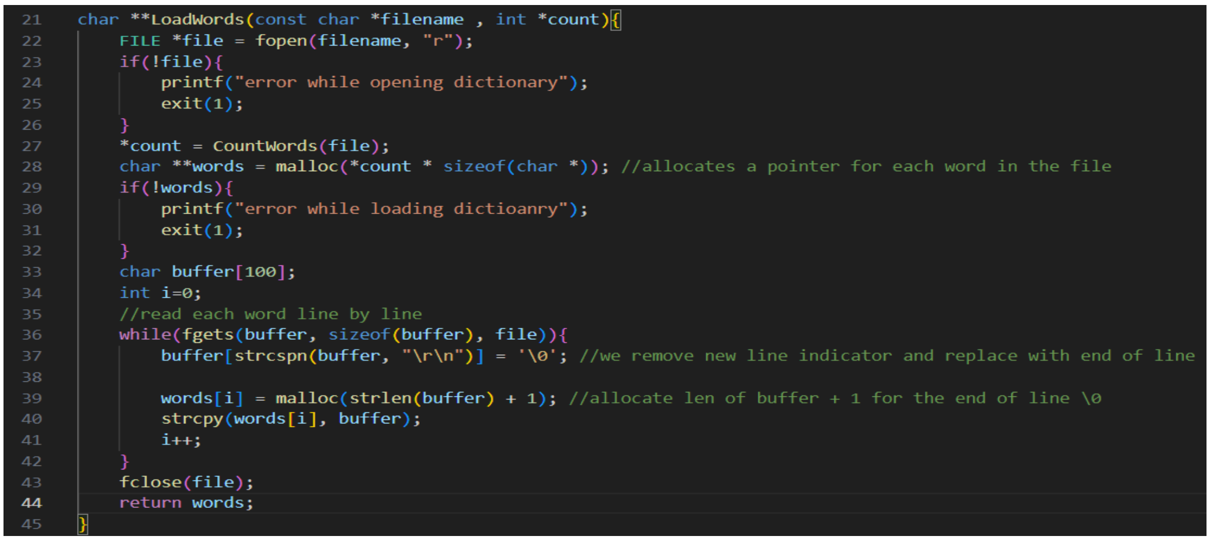
we'll start by an introduction to the utils.c functions used in game.c:

1) the function **countWords** will read a file and counts how many words in it by incrementing a counter for each line read.

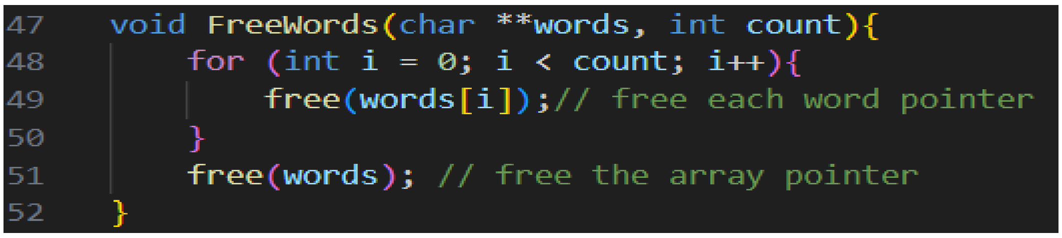


```
int countWords(FILE *file){
    char buffer[100];
    int counter = 0;
    while (fgets(buffer, sizeof(buffer), file)){
        counter++;
    }
    rewind(file);
    return counter;
}
```

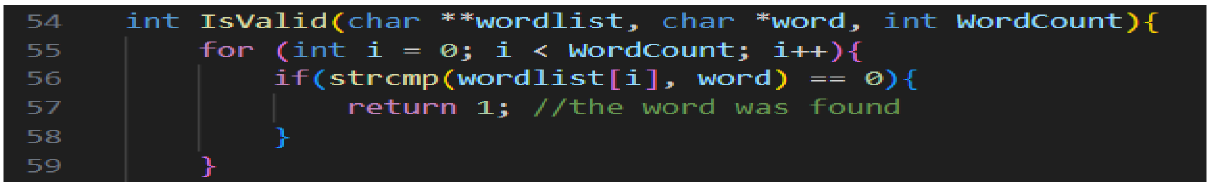
2) the function **loadwords** will load the words contained inside the words.txt files into a dynamic array. this improves the spatial complexity of the code and avoids using a static array. it uses fopen to open the file and then uses fgets to read each line and store into a buffer that serves as a middleman between the file and the words dynamic array where we store each word.


```c
char **LoadWords(const char *filename , int *count){
    FILE *file = fopen(filename, "r");
    if(!file){
        printf("error while opening dictionary");
        exit(1);
    }
    *count = CountWords(file);
    char **words = malloc(*count * sizeof(char *)); //allocates a pointer for each word in the file
    if(!words){
        printf("error while loading dictioanry");
        exit(1);
    }
    char buffer[100];
    int i=0;
    //read each word line by line
    while(fgets(buffer, sizeof(buffer), file)){
        buffer[strcspn(buffer, "\r\n")] = '\0'; //we remove new line indicator and replace with end of line

        words[i] = malloc(strlen(buffer) + 1); //allocate len of buffer + 1 for the end of line \0
        strcpy(words[i], buffer);
        i++;
    }
    fclose(file);
    return words;
}
```

3) The function **FreeWords** will be used to free the dynamic array after we are done with it.

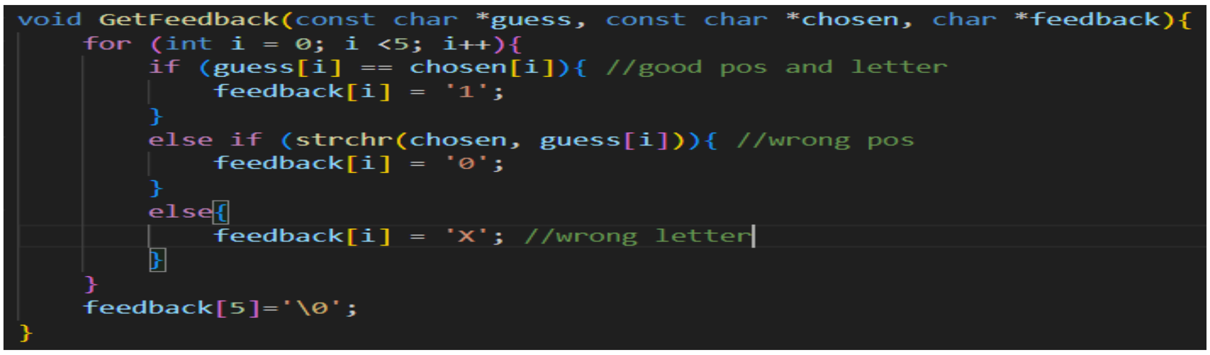

```c
void FreeWords(char **words, int count){
    for (int i = 0; i < count; i++){
        free(words[i]);// free each word pointer
    }
    free(words); // free the array pointer
}
```

4) the function **isValid** check if the user input of the guess is a valid word contained in the words.txt file by comparing the input with each word in the file using strcmp.


```c
int IsValid(char **wordlist, char *word, int WordCount){
    for (int i = 0; i < WordCount; i++){
        if(strcmp(wordlist[i], word) == 0){
            return 1; //the word was found
        }
    }
```

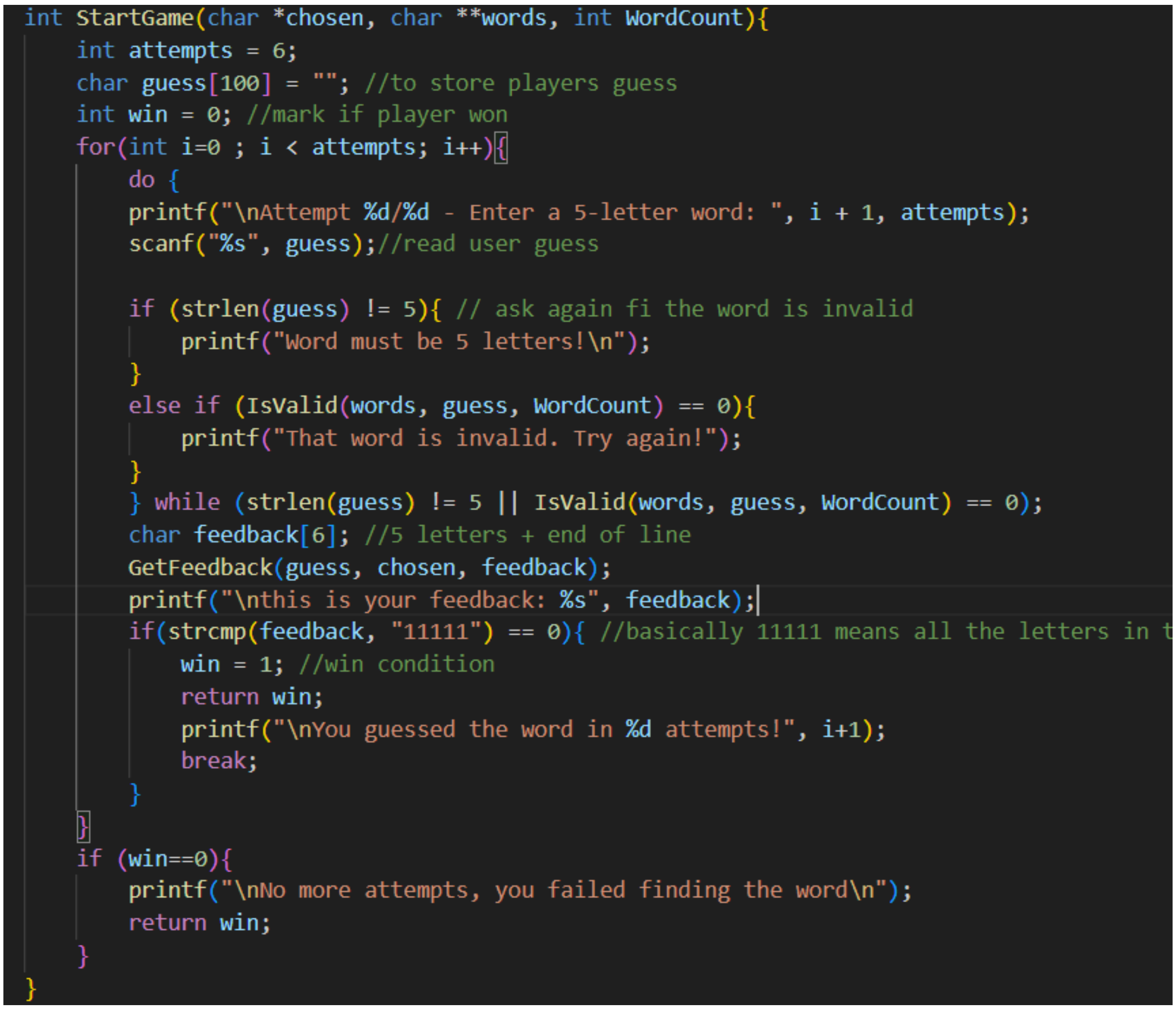
Next, we move to the game logic:

I) In the game.c file. we use a **GetFeedback** function that will analyse the user guess and returns a corresponding feedback. Since we cannot use colors in a command line interface, we will give each case a letter instead. If the letter exists in the word and is in the right spot, we return '1', if the letter exists in the word but is in the wrong spot, we return '0' and if the letter does not exist in the word, we return 'X' symbolising that the letter is wrong. the function takes the user guess then returns an array that contains the sequence of 0 , 1 and X as feedback



```c
void GetFeedback(const char *guess, const char *chosen, char *feedback){
    for (int i = 0; i <5; i++){
        if (guess[i] == chosen[i]){ //good pos and letter
            feedback[i] = '1';
        }
        else if (strchr(chosen, guess[i])){ //wrong pos
            feedback[i] = '0';
        }
        else{
            feedback[i] = 'X'; //wrong letter
        }
    }
    feedback[5]='\0';
}
```

II) The **<u>startGame</u>** function is where we use all the previously mentioned functions. For a starter, we need an attempts variable to determine how many guesses the user has, a guess variable to store the user input and variable to flag the win condition.the loop will ask for the user input, checks if it's valid the returns a feedback. if the feedback is "11111" sets the win flag to 1 and exits else it redoes this 5 more times. if the win flag is still set to zero by the 6th guess, the user fails the game.

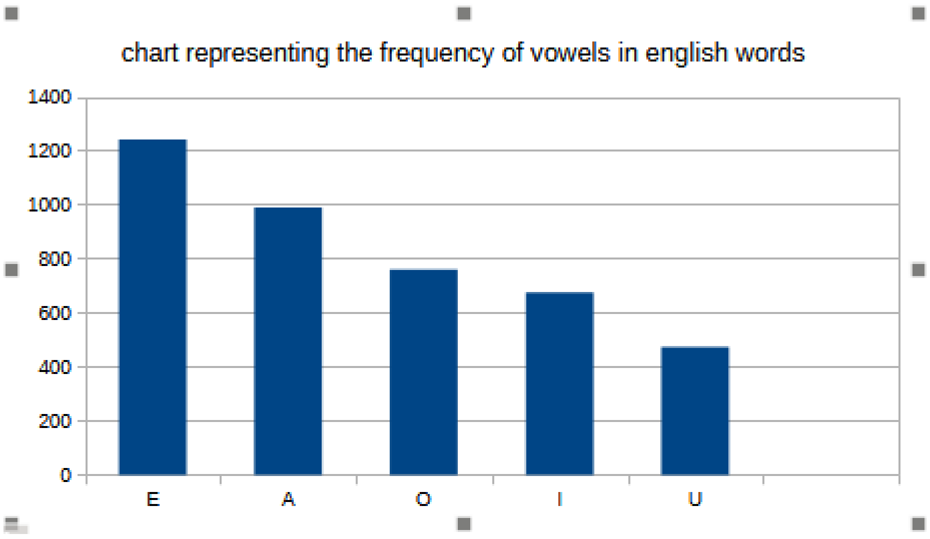


```c
int StartGame(char *chosen, char **words, int WordCount){
    int attempts = 6;
    char guess[100] = ""; //to store players guess
    int win = 0; //mark if player won
    for(int i=0 ; i < attempts; i++){
        do {
        printf("\nAttempt %d/%d - Enter a 5-letter word: ", i + 1, attempts);
        scanf("%s", guess);//read user guess

        if (strlen(guess) != 5){ // ask again fi the word is invalid
            printf("Word must be 5 letters!\n");
        }
        else if (IsValid(words, guess, WordCount) == 0){
            printf("That word is invalid. Try again!");
        }
        } while (strlen(guess) != 5 || IsValid(words, guess, WordCount) == 0);
        char feedback[6]; //5 letters + end of line
        GetFeedback(guess, chosen, feedback);
        printf("\nthis is your feedback: %s", feedback);
        if(strcmp(feedback, "11111") == 0){ //basically 11111 means all the letters in t
            win = 1; //win condition
            return win;
            printf("\nYou guessed the word in %d attempts!", i+1);
            break;
        }
    }
    if (win==0){
        printf("\nNo more attempts, you failed finding the word\n");
        return win;
    }
}
```

Statistical comparaison

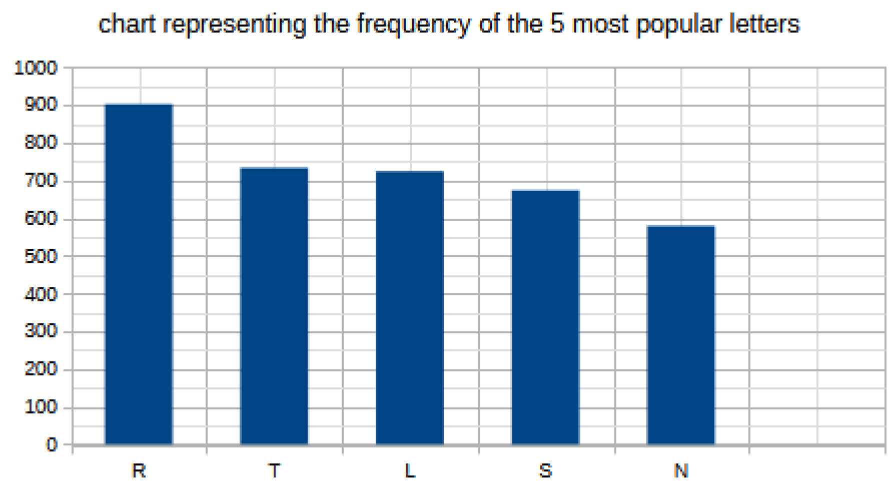
Our game solver has 6 attempts to guess the right word randomly from the list of words that contains more than two thousands words. Our goal is to make it solve the game in the least possible number of guesses.

In order to do that we must take a look at the most used letters and vowels in the english language and help our solver start with the right words that will narrow the list as much as possible and reduce the number of guesses.



This chart shows us that the letter E is the most frequent vowel in the english 5 letters words followed by the letter A then O and last is U.

This means that our random target word is likely to have the vowel E and A in it



this chart shows that R and T are the most used letters in the 5 letters english words. That means the target word will likely have them

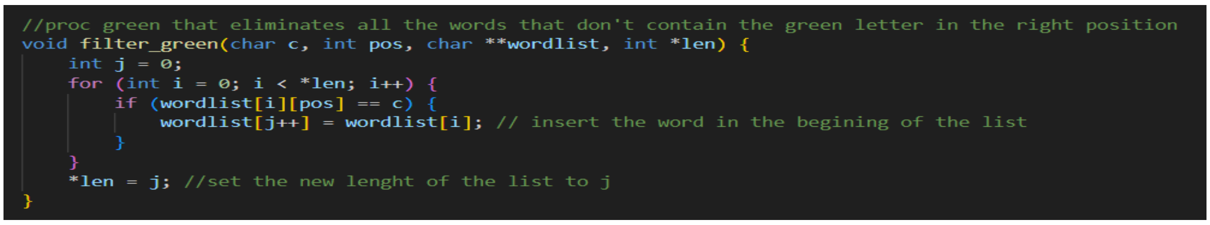
By combining the data from the two charts we get to the conclusion that we must pre define two words as a default first two guesses for the solver

these words will be **SNORT** and **ADIEU**. They cover all the vowels and totally different letter and are more likely to narrow the list better than all other possible words

explanation of the solver logic

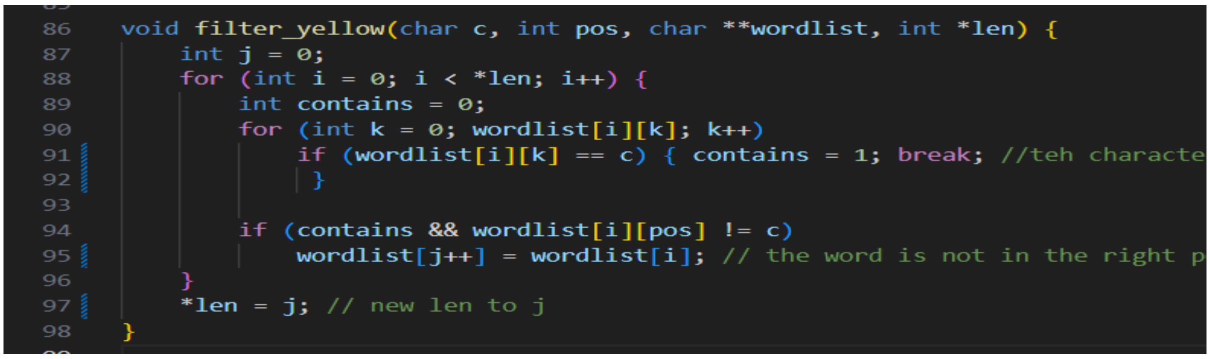
1) The function **filter_green** will take the wordlist, a position, and a character. It will search for all the words that contain the exact character in the exact given position. then narrows the wordlist by eliminating the words that don't meet that condition.

the narrowing is done by inserting those words at the start of the list.



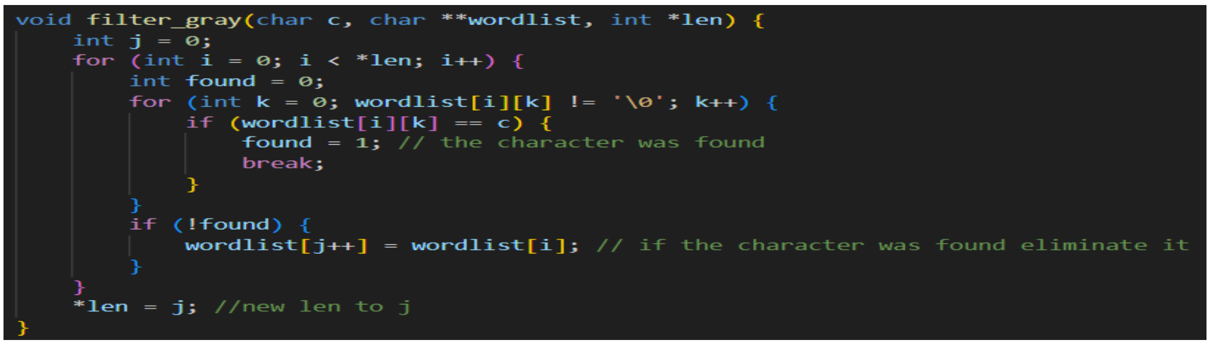
```c
//proc green that eliminates all the words that don't contain the green letter in the right position
void filter_green(char c, int pos, char **wordlist, int *len) {
    int j = 0;
    for (int i = 0; i < *len; i++) {
        if (wordlist[i][pos] == c) {
            wordlist[j++] = wordlist[i]; // insert the word in the begining of the list
        }
    }
    *len = j; //set the new lenght of the list to j
}
```

2) the function **filter_yellow** will also take the wordlist , a position and the char then narrows the list by eliminating all the words that don't contain the exact character in any position.



```c
void filter_yellow(char c, int pos, char **wordlist, int *len) {
    int j = 0;
    for (int i = 0; i < *len; i++) {
        int contains = 0;
        for (int k = 0; wordlist[i][k]; k++)
            if (wordlist[i][k] == c) { contains = 1; break; //teh characte
            }

        if (contains && wordlist[i][pos] != c)
            wordlist[j++] = wordlist[i]; // the word is not in the right p
    }
    *len = j; // new len to j
}
```

3) the function **filter_gray** will take the wordlist, and character c then eliminate all the words that contain that character.

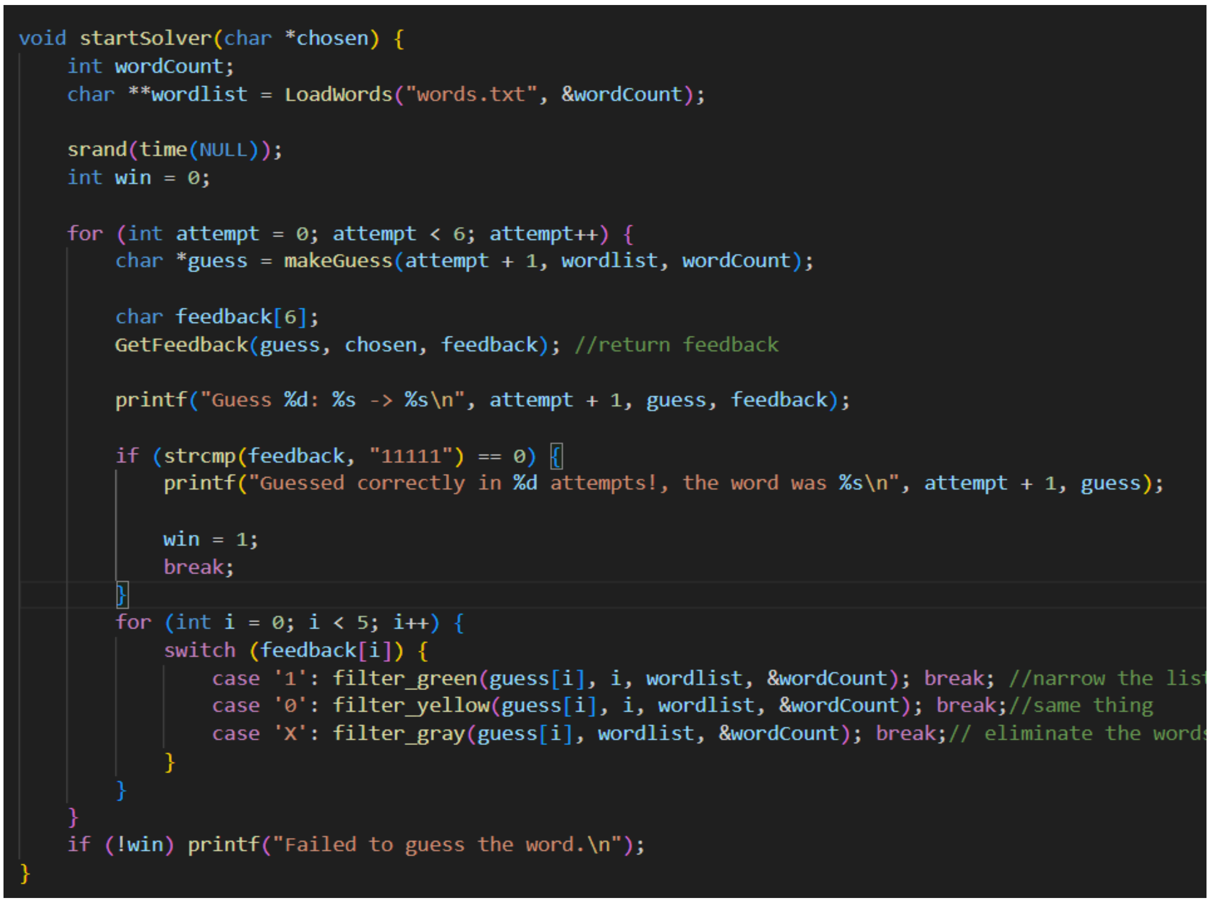


```c
void filter_gray(char c, char **wordlist, int *len) {
    int j = 0;
    for (int i = 0; i < *len; i++) {
        int found = 0;
        for (int k = 0; wordlist[i][k] != '\0'; k++) {
            if (wordlist[i][k] == c) {
                found = 1; // the character was found
                break;
            }
        }
        if (!found) {
            wordlist[j++] = wordlist[i]; // if the character was found eliminate it
        }
    }
    *len = j; //new len to j
}
```

Next we move the solver file:

1) the **startSolver** function is where we use the previously mentionned functions. First of all it loads the words into a wordlist then enters a loop, We store the makeGuess return in a guess variable then get the feedback. after getting the feedback we browse it character by character and invoke the filter functions. if the char is '1' we call the filter_green function to save the corresponding words in the list, goes the same way for the filter_yellow and filter_gray functions.

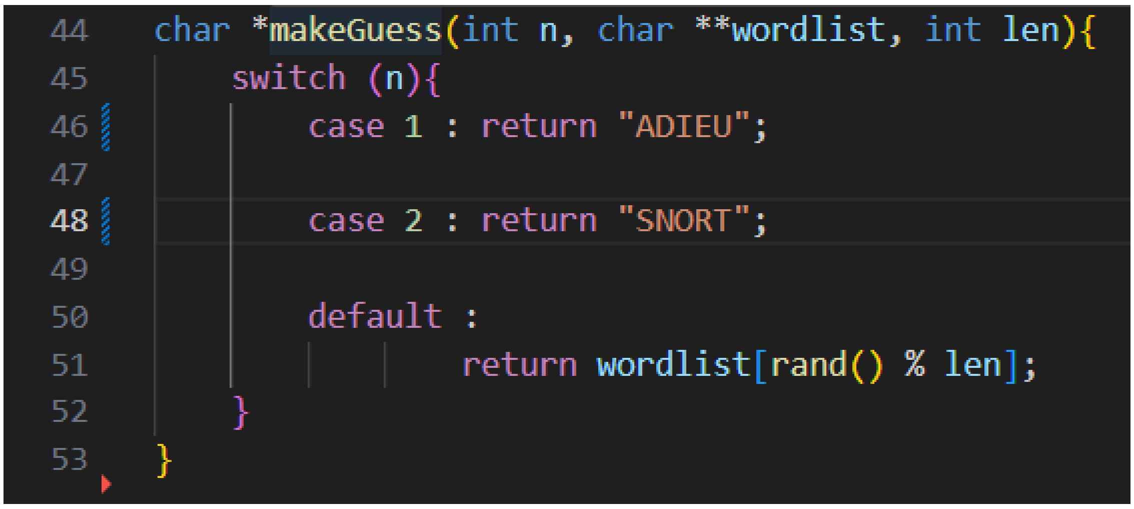
if the feedback is "11111" sets the win flag to 1 and exits signaling that the solver finished solving.



```c
void startSolver(char *chosen) {
    int wordCount;
    char **wordlist = LoadWords("words.txt", &wordCount);

    srand(time(NULL));
    int win = 0;

    for (int attempt = 0; attempt < 6; attempt++) {
        char *guess = makeGuess(attempt + 1, wordlist, wordCount);

        char feedback[6];
        GetFeedback(guess, chosen, feedback); //return feedback

        printf("Guess %d: %s -> %s\n", attempt + 1, guess, feedback);

        if (strcmp(feedback, "11111") == 0) {
            printf("Guessed correctly in %d attempts!, the word was %s\n", attempt + 1, guess);

            win = 1;
            break;
        }
        for (int i = 0; i < 5; i++) {
            switch (feedback[i]) {
                case '1': filter_green(guess[i], i, wordlist, &wordCount); break; //narrow the lis
                case '0': filter_yellow(guess[i], i, wordlist, &wordCount); break;//same thing
                case 'X': filter_gray(guess[i], wordlist, &wordCount); break;// eliminate the word
            }
        }
    }
    if (!win) printf("Failed to guess the word.\n");
}
```

2) the function **makeGuess** returns a random word from the words list by using rand(), however it always forces the first 2

guesses to be ADIEU and SNORT as they have the best performance



```c
char *makeGuess(int n, char **wordlist, int len){
    switch (n){
        case 1 : return "ADIEU";

        case 2 : return "SNORT";

        default :
                return wordlist[rand() % len];
    }
}
```