WORDLE GAME

Made by (isil C):

- ♡ Ramoul Meriem
- ♡ Seray Imene
- ♡ Dekrah Lakeha

WHAT IS THE WORDLE GAME :

Wordle is a word-guessing game where the player must find a hidden word within a limited number of attempts. Each guess gives feedback: correct letters in the correct position, correct letters in the wrong position, or letters that do not appear in the word at all.

THE GAME FUNCTIONS :

- ♡ loaddictionary
- ♡ pickrandomword
- ♡ isvalidguess
- ♡ getfeedback
- ♡ displayfeedback
- ♡ main

GAME EXECUTION OVERVIEW :

The Wordle game is a word-guessing game in which the computer first selects a hidden word randomly from a predefined dictionary. The player is then invited to repeatedly enter guesses of the same length as the hidden word, with a limited number of attempts. After each guess, the game compares the user's input with the secret word letter by letter and provides feedback to guide the next attempt. For every position, if a guessed letter matches the corresponding letter in the hidden word, it is marked as correct and well-placed; if the letter exists in the hidden word but appears in a different position, it is marked as correct but misplaced; otherwise, the letter is marked as absent from the word. The game continues until the player successfully discovers the hidden word, at which point the game ends and displays a success message; otherwise, the secret word is revealed at the end of the game. The Wordle game component is responsible for managing the secret word selection, handling user input, validating guesses, generating feedback, and determining the termination condition.

1/loadDictionary

```
#include <stdio.h>
#include "wordle.h"

int loadDictionary(const char *filename, char dictionary[][WORD_BUFFER]) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        printf("Error opening file %s\n", filename);
        return 0;
    }

    int count = 0;
    while (count < MAX_WORDS && fscanf(file, "%5s", dictionary[count]) == 1) {
        count++;
    }

    fclose(file);
    return count;
}
```

THE GOAL OF THE FUNCTION

```
The function reads words from the file
and stores them into a 2D array (dictionary).
It returns the number of words successfully loaded.
```

`<stdio.h>` basic used libary allows us to use the functions `printf`, `fopen`, `fclose`, and `fscanf`. `"wordle.h"` is our own header file that contains things like `WORD_BUFFER` and `MAX_WORDS`

Function Parameters

- `const char *filename`: the name of the dictionary file we want to load.
- `dictionary[][WORD_BUFFER]`: a 2D array where each row represents one word.

Opening the File

```
FILE *file = fopen(filename, "r");
```

The file is opened in read mode ”r”.

```
if (!file) {
       printf("Error opening file %s\n", filename);
       return 0;
```

If read mode fails `file` becomes `NULL` and we return 0.

Reading Words

We start counting from zero:

```c
int count = 0;
```

Then read one word at a time using:

```c
while (count < MAX_WORDS && fscanf(file, "%5s", dictionary[count]) == 1) {
    count++;
}
```

- we use the while loop to make sure that the count is smaller than the "MAX-WORDS "and to read a word from the file
- `"%5s"` reads up to 5 characters (letters) from the file
- each word is placed in `dictionary[count]`
- ==1 means it successfully read one string if EOF or the or reading failed
- if the read is successful, `count++`

Closing the File

After reading all words, close the file to free resources and we use :

```c
fclose(file);
```

Return Value

The function returns the number of words successfully loaded

```c
return count;
```

COMPLEXITY :

```
complixity of this function is --->O(N)
```

2/pick_random_word

Code:

```c
#include <stdlib.h>
#include <time.h>
#include <string.h>

char *pick_random_word(char **dictionary, int size) {
    if (size <= 0 || dictionary == NULL) {
        return NULL;
    }

    static int initialized = 0;
    if (!initialized) {
        srand(time(NULL));
        initialized = 1;
    }

    int index = rand() % size;
    char *word = dictionary[index];

    char *copy = malloc(strlen(word) + 1);
    if (copy == NULL) {
        return NULL;
    }

    strcpy(copy, word);
    return copy;
}
```

Explanation:

Used Libraries in the function:

- `stdlib.h`: allows us to use `rand()`, `srand()`, and `malloc()`.
- `time.h`: gives access to `time(NULL)` for randomness seeding.
- `string.h`: used for `strlen()` and `strcpy()`.

Function Header

```c
char *pick_random_word(char **dictionary, int size)
```

Returns a pointer to a dynamically allocated string. Receives an array of words and the number of words.

Input Validation

```c
if (size <= 0 || dictionary == NULL) {
    return NULL;
}
```

this condition verify If the dictionary is empty or invalid, the function stops immediately.

Static Initialization

```c
static int initialized = 0;
if (!initialized) {
    srand(time(NULL));
    initialized = 1;
}
```

A static variable ensures that `srand()` is executed only once. This avoids repeating the random seed unnecessarily.

Random Index Selection

```
int index = rand() % size;
```

Generates a random number between `0` and `size - 1`.

Selecting the Word

```
char *word = dictionary[index];
```

Picks a random string from the dictionary.

Memory Allocation

```
char *copy = malloc(strlen(word) + 1);
```

Allocates memory for the copy of the word (`+1` for \0 means the end of the string).

Allocation Check

```
if (copy == NULL) {
    return NULL;
}
```

Ensures that `malloc()` succeeded else return null

Copying the Word

```
strcpy(copy, word);
```

Copies the selected word into the newly allocated memory.

Returning the Result

```
return copy;
```

Returns the random word's independent copy.

COMPLEXITY:

```
The \verb|pick_random_word| function has a time complexity of \textbf{O(L)}
where L is the length of the selected word
```

3/IsValidGuess

Code

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "wordle.h"

// convert uppercase letters in a string to lowercase in-place
void tolowerCase(char str[]){
    for (int i = 0; str[i] != '\0'; i++){
        if (isupper((unsigned char)str[i])){
            str[i] = tolower((unsigned char)str[i]);
        }
    }
}

// Validate guess: must be WORD_LEN letters and exist in dictionary
int isValidGuess(char guess[], char dictionary[][WORD_BUFFER], int numWords) {
    tolowerCase(guess);

    if (strlen(guess) != WORD_LEN) {
        return 0;
    }

    for (int i = 0; i < WORD_LEN; i++) {
        if (!isalpha((unsigned char)guess[i])) {
            return 0;
        }
    }

    for (int i = 0; i < numWords; i++) {
        if (strcmp(guess, dictionary[i]) == 0) {
            return 1;
        }
    }

    return 0;
}
```

Explanation

Included Libraries

- `stdio.h`: standard input/output functions (included in most C projects).
- `string.h`: used for:

- `strlen(): to check the guess length`
- `strcmp()` : to compare guess with dictionary words
- `ctype.h`: provides:
- `isupper(): to check uppercase letters`
- `tolower()`: convert letters to lowercase
- `isalpha(): verify characters are letters only`
- `"wordle.h"`: contains project constants such as `WORD-LEN` and `WORD-BUFFER`.

tolowerCase Function

```
void tolowerCase(char str[]){
    for (int i = 0; str[i] != '\0'; i++){
        if (isupper((unsigned char)str[i])){
            str[i] = tolower((unsigned char)str[i]);
        }
    }
}
```

```
  for (int i = 0; str[i] != '\0'; i++)
```

this for loop keeps running as long as `str[i]` is **not the null terminator** This ensures we process **every character in the string**, stopping only at the end.

```
 if (isupper((unsigned char)str[i])){
            str[i] = tolower((unsigned char)str[i]);
        }
    }
}
```

this one converts all uppercase letters in the input string to lowercase cause our dictionary is in lowercase characters

Checks if each character is uppercase using `isupper()`. - Converts uppercase letters with `tolower()` and stores them back in the same string.

isValidGuess

```
int isValidGuess(char guess[], char dictionary[][WORD_BUFFER], int numWords) {
    tolowerCase(guess);

    if (strlen(guess) != WORD_LEN) {
        return 0;
    }

    for (int i = 0; i < WORD_LEN; i++) {
        if (!isalpha((unsigned char)guess[i])) {
            return 0;
        }
    }

    for (int i = 0; i < numWords; i++) {
        if (strcmp(guess, dictionary[i]) == 0) {
            return 1;
        }
    }

    return 0;
}
```

Function Header

```
int isValidGuess(char guess[], char dictionary[][WORD_BUFFER], int numWords)
```

Receives a guess string from the user , the dictionary 2D array, and number of words in the dictionary. Returns 1 if the guess is valid, 0 otherwise.

Convert Guess to Lowercase

```
tolowerCase(guess);
```

Ensures the guess is lowercase for case-insensitive comparison with dictionary words to void any problems.

Check Length

```
if (strlen(guess) != WORD_LEN) {
    return 0;
}
```

compares the length of the guess word with the WORD-LENGHT and rejects the guess if it does not have exactly `WORD-LEN` letters (5+/0).

Check Alphabetic Characters

```
for (int i = 0; i < WORD_LEN; i++) {
    if (!isalpha((unsigned char)guess[i])) {
        return 0;
    }
}
```

isalpha()verifies that all characters are letters not symbols or numbers just english letters ; any invalid character makes the guess invalid and return 0.

Check Dictionary

```
for (int i = 0; i < numWords; i++) {
    if (strcmp(guess, dictionary[i]) == 0) {
        return 1;
    }
}
```

strcmp() compares the guess with each word in the dictionary. If an exact match is found → guess is valid .

Return Invalid

```
return 0;
```

If no match is found → the guess is invalid.

COMPLEXITY:

```
The \verb|isValidGuess| function has a time complexity of \textbf{O(N × L)}
where N is the size of the dictionary and L is the word length
```

4/getFeedback

Code:

```
#include <stdio.h>
#include <string.h>
#include "wordle.h"

void getFeedback(const char *secret, const char *guess, char *feedback) {
    int len = strlen(secret);
    int used[WORD_LEN]; // mark letters used for yellow

    for (int i = 0; i < len; i++) used[i] = 0;

    // First pass: mark Greens
    for (int i = 0; i < len; i++) {
        if (guess[i] == secret[i]) {
            feedback[i] = 'G';
            used[i] = 1;
        } else {
            feedback[i] = 'B';
        }
    }

    // Second pass: mark Yellows
    for (int i = 0; i < len; i++) {
        if (feedback[i] == 'G') continue;

        for (int j = 0; j < len; j++) {
            if (!used[j] && guess[i] == secret[j]) {
                feedback[i] = 'Y';
                used[j] = 1;
                break;
            }
        }
    }

    feedback[len] = '\0';
}
```

THE GOAL OF THE FUNCTION

```
This function compares the player's guess with the secret word.
It gives feedback for each letter:
G = correct letter, correct position
Y = correct letter, wrong position
B = letter does not appear in the secret word
```

Used libaries:

`<stdio.h>` basic library used for debugging (printf if needed).

`<string.h>` used for `strlen()` to get the word length.

`"wordle.h"` contains constants like `WORD-LEN`.

Function Parameters

```
void getFeedback(const char *secret, const char *guess, char *feedback)
```

- `const char *secret`: the real secret chosen word.
- `const char *guess`: the player's attempt.
- `char *feedback`: an array where we write G / Y / B.

Getting the Word Length

```
int len = strlen(secret);
```

We measure the secret word length to know how many letters to compare.

Preparing the `used[]` Array

```
int used[WORD_LEN];
for (int i = 0; i < len; i++) used[i] = 0;
```

- `used[]` helps mark which letters in the secret word are already matched.
- It is very important for correctly handling repeated letters.
- We initialize all values to 0 → meaning "not used yet".

First Pass: Marking Greens

```
for (int i = 0; i < len; i++) {
    if (guess[i] == secret[i]) {
        feedback[i] = 'G';
        used[i] = 1;
    } else {
        feedback[i] = 'B';
    }
}
```

- We compare each letter in the guess with the same position in the secret word.
- If they match → mark **Green** ('G').
- Mark the letter as used so it won't be used again for Yellow.
- Otherwise → temporarily mark as **Black** ('B').

Second Pass: Marking Yellows

```
for (int i = 0; i < len; i++) {
    if (feedback[i] == 'G') continue;

    for (int j = 0; j < len; j++) {
        if (!used[j] && guess[i] == secret[j]) {
            feedback[i] = 'Y';
            used[j] = 1;
            break;
        }
    }
}
```

- Skip Greens → they are already correct.
- For each non-green letter:
  - search the secret word for the same letter
  - only use positions not already taken (used[j] == 0)
- If found → mark the guess letter as **Yellow**.
- Stop after first match to avoid marking multiple Yellows for a single letter.

Null-Terminating the Feedback

```
feedback[len] = '\0';
```

Adds the end-of-string marker so the feedback can be printed normally.
COMPLEXITY:

```
The \verb|getFeedback| function has a time complexity of \textbf{O(L²)},
```

5/display_feedback

```c
#include <stdio.h>
#include "wordle.h"

/* Print a single letter in a colored box according to feedback:
 *  G -> green background
 *  Y -> yellow background
 *  B -> gray background
 * Uses black text on colored background for readability.
 */
static void print_colored_box(char f, char c) {
    if (f == 'G') {
        /* black text on green background */
        printf("\033[30;42m %c \033[0m ", c);
    } else if (f == 'Y') {
        /* black text on yellow background */
        printf("\033[30;43m %c \033[0m ", c);
    } else {
        /* black text on bright black (gray) background */
        printf("\033[37;100m %c \033[0m ", c);
    }
}

void display_feedback(const char *feedback, const char *guess) {
    for (int i = 0; i < WORD_LEN; i++) {
        char c = guess[i];
        char f = feedback[i];
        /* If guess or feedback shorter, print placeholder */
        if (c == '\0') {
            printf("    ");
        } else {
            print_colored_box(f, c);
        }
    }
}
```

THE GOAL OF THE FUNCTIONS

```
These functions display the Wordle feedback for a player's guess.
Each letter is shown inside a colored box according to the feedback:
G = correct letter and correct position (green)
Y = correct letter but wrong position (yellow)
B = letter does not appear in the secret word (gray)
```

`<stdio.h>`: provides `printf()` to print letters and colored boxes. `"wordle.h"`: contains game constants like `WORD_LEN`.

Function Parameters

- `const char *feedback`: array of feedback characters for each letter ('G','Y','B').
- `const char *guess`: the player's guessed word.

Helper Function: `print_colored_box`

```
static void print_colored_box(char f, char c)
```

- Declared `static` → private to this file.
- Parameters: `f` = feedback character, `c` = letter to display.
- Responsible for printing a single letter inside a colored box according to feedback.

Printing Colored Boxes with ANSI Codes

```
if (f == 'G') {
    printf("\033[30;42m %c \033[0m ", c);
} else if (f == 'Y') {
    printf("\033[30;43m %c \033[0m ", c);
} else {
    printf("\033[37;100m %c \033[0m ", c);
}
```

- Uses ANSI escape codes to color text in the terminal.
- `30` = black text, `42` = green background.
- `43` = yellow background.
- `37;100` = white text on bright black (gray) background.
- \\`033[0m` resets colors after printing each letter.

Main Function: `display_feedback`

```
for (int i = 0; i < WORD_LEN; i++) {
    char c = guess[i];
    char f = feedback[i];
    if (c == '\0') {
        printf("   ");
    } else {
        print_colored_box(f, c);
    }
}
```

- Loops over each letter in the guess word (`WORD_LEN` iterations).
- Copies guess letter into `c` and feedback into `f`.
- If guess is shorter than expected, prints empty spaces for alignment.

- Otherwise, calls `print_colored_box(f,c)` to print the letter with the correct color.
- Ensures the feedback output visually matches the Wordle style.

COMPLEXITY:

```
The \verb|main| function has a time complexity of \textbf{O(N × L)}, where N is the number of words
```

6/ Main function

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "wordle.h"

int main() {
    char dictionary[MAX_WORDS][WORD_BUFFER];
    char *dict_ptrs[MAX_WORDS];  // Array of pointers for pick_random_word
    char feedback[WORD_BUFFER];
    char guess[WORD_BUFFER];
    char *secret_word;
    int num_words;
    int attempts = 0;
    int max_attempts = 6;
    int won = 0;

    // Load dictionary from file
    num_words = loadDictionary("word.txt", dictionary);
    if (num_words == 0) {
        printf("\033[1;91mError! Could not load dictionary.\n\033[0m");
        return 1;
    }

    // Create array of pointers to dictionary words
    for (int i = 0; i < num_words; i++) {
        dict_ptrs[i] = dictionary[i];
    }

    /* Welcome (pink) and one-time legend */
    printf("\033[1;35m=== Welcome to Wordle! ===\n\033[0m");
    printf("You have %d attempts to guess the %d-letter word.\n\n", max_attempts, WORD_LEN);
    printf("Legend (shown once): (\033[1;32m Green\033[0m = correct position, \033[1;33m Yellow\033[0m = wrong

    /* persistent small board to show past guesses */
    char past_guesses[6][WORD_BUFFER];
    char past_feedbacks[6][WORD_BUFFER];
    for (int i = 0; i < max_attempts; i++) {
        past_guesses[i][0] = '\0';
        past_feedbacks[i][0] = '\0';
    }
    int printed_legend = 1; /* we already printed it above */
    /* keyboard state: 'U' unknown, 'G' green, 'Y' yellow, 'B' gray */
    char keyboard_state[26];
    for (int i = 0; i < 26; ++i) keyboard_state[i] = 'U';

    // Pick a random word
    secret_word = pick_random_word(dict_ptrs, num_words);
    if (secret_word == NULL) {
        printf("\033[1;91mError! Could not select a word.\n\033[0m");
        return 1;
    }

    // Game loop
    while (attempts < max_attempts && !won) {
        /* Clear screen and re-render compact UI (legend shown only once) */
        printf("\033[2J\033[H");
        /* header */
        printf("\033[1;35m=== Wordle ===\033[0m\n");
        printf("Attempts: %d/%d\n", attempts, max_attempts);
        if (printed_legend) {
            /* show brief note that legend was shown once */
            printf("(Legend shown at start)\n\n");
        } else {
            printf("\n");
        }

        /* Render board */
        printf("Previous guesses:\n");
        for (int i = 0; i < max_attempts; i++) {
            printf("%d: ", i + 1);
            if (past_guesses[i][0] == '\0') {
                for (int k = 0; k < WORD_LEN; k++) printf("[   ] ");
                printf("\n");
            } else {
                display_feedback(past_feedbacks[i], past_guesses[i]);
                printf("\n");
            }
        }

        /* Render keyboard (QWERTY) */
        const char *row1 = "qwertyuiop";
        const char *row2 = "asdfghjkl";
        const char *row3 = "zxcvbnm";
        printf("\nKeyboard:\n");
        for (int r = 0; row1[r]; ++r) {
```

```
        char ch = row1[r];
        char s = keyboard_state[ch - 'a'];
        if (s == 'G') printf("\033[30;42m %c \033[0m ", ch);
        else if (s == 'Y') printf("\033[30;43m %c \033[0m ", ch);
        else if (s == 'B') printf("\033[37;100m %c \033[0m ", ch);
        else printf("[ %c ] ", ch);
    }
    printf("\n");
    for (int r = 0; row2[r]; ++r) {
        char ch = row2[r];
        char s = keyboard_state[ch - 'a'];
        if (s == 'G') printf("\033[30;42m %c \033[0m ", ch);
        else if (s == 'Y') printf("\033[30;43m %c \033[0m ", ch);
        else if (s == 'B') printf("\033[37;100m %c \033[0m ", ch);
        else printf("[ %c ] ", ch);
    }
    printf("\n");
    for (int r = 0; row3[r]; ++r) {
        char ch = row3[r];
        char s = keyboard_state[ch - 'a'];
        if (s == 'G') printf("\033[30;42m %c \033[0m ", ch);
        else if (s == 'Y') printf("\033[30;43m %c \033[0m ", ch);
        else if (s == 'B') printf("\033[37;100m %c \033[0m ", ch);
        else printf("[ %c ] ", ch);
    }
    printf("\n\n");

    printf("\nEnter your guess (attempt %d/%d): ", attempts + 1, max_attempts);

    // Get user input
    if (fgets(guess, sizeof(guess), stdin) == NULL) {
        printf("\033[1;91mError reading input.\n\033[0m");
        continue;
    }

    // Remove newline character if present
    if (guess[strlen(guess) - 1] == '\n') {
        guess[strlen(guess) - 1] = '\0';
    }

    // Validate the guess
    if (!isValidGuess(guess, dictionary, num_words)) {
        printf("\033[1;91mError! Invalid guess. Must be a 5-letter word in the dictionary.\n\033[0m\n");
        continue;
    }

    // Get feedback
    getFeedback(secret_word, guess, feedback);
```

```
        // Store and show in board
        strncpy(past_guesses[attempts], guess, WORD_BUFFER - 1);
        past_guesses[attempts][WORD_BUFFER - 1] = '\0';
        strncpy(past_feedbacks[attempts], feedback, WORD_BUFFER - 1);
        past_feedbacks[attempts][WORD_BUFFER - 1] = '\0';

        /* Update keyboard state: precedence G > Y > B */
        for (int i = 0; i < WORD_LEN; ++i) {
            char c = guess[i];
            if (c < 'a' || c > 'z') continue;
            int idx = c - 'a';
            char f = feedback[i];
            if (f == 'G') keyboard_state[idx] = 'G';
            else if (f == 'Y') {
                if (keyboard_state[idx] != 'G') keyboard_state[idx] = 'Y';
            } else { /* B */
                if (keyboard_state[idx] != 'G' && keyboard_state[idx] != 'Y') keyboard_state[idx] = 'B';
            }
        }

        // Display feedback for this guess
        printf("Feedback: ");
        display_feedback(feedback, guess);
        printf("\n\n");

        // Check if the guess is correct
        if (strcmp(guess, secret_word) == 0) {
            printf("\033[1;32m Congratulations! You guessed the word: %s\n\033[0m", secret_word);
            won = 1;
        } else {
            attempts++;
        }
    }

    // Game over
    if (!won) {
        printf("\033[1;91m Game Over! The word was: %s\n\033[0m", secret_word);
    }

    // Free allocated memory
    free(secret_word);

    return 0;
}
```

Included libraries (code)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "wordle.h"
```

explanation:

- `<stdio.h>` — input/output functions such as `printf()` and `fgets()` used for the game UI and reading input.
- `<stdlib.h>` — dynamic memory and utilities: `malloc()`, `free()` and other helpers the program uses (e.g., `exit()` if needed).
- `<string.h>` — string handling: `strlen()`, `strcmp()`, `strncpy()` used for validating and copying guesses/feedback.
- `"wordle.h"` — your project header with constants and prototypes: `WORD_LEN`, `WORD_BUFFER`, `MAX_WORDS`, and declarations for helper functions (`loadDictionary`, `pick_random_word`, `getFeedback`, `display_feedback`, `isValidGuess`).

1- Variable declarations

```c
char dictionary[MAX_WORDS][WORD_BUFFER];
char *dict_ptrs[MAX_WORDS];  // Array of pointers for pick_random_word
char feedback[WORD_BUFFER];
char guess[WORD_BUFFER];
char *secret_word;
int num_words;
int attempts = 0;
int max_attempts = 6;
int won = 0;
```

What each variable does (brief):

- `dictionary[][]` — 2D array that stores all words loaded from file (each row = one word).
- `dict_ptrs[]` — array of `char*` that point to rows in `dictionary`; used because `pick_random_word` expects a pointer-array.
- `feedback[]` — buffer to hold the feedback string (e.g., `"GYBBG"`).
- `guess[]` — user input buffer for the guessed word.
- `secret_word` — pointer to dynamically allocated string returned by `pick_random_word`.
- `num_words` — number of words successfully loaded.
- `attempts`, `max_attempts` — track tries (Wordle uses 6).
- `won` — flag (0/1) whether player guessed correctly.

2- Load dictionary

```c
// Load dictionary from file
num_words = loadDictionary("word.txt", dictionary);
if (num_words == 0) {
    printf("\033[1;91mError! Could not load dictionary.\n\033[0m");
    return 1;
}
```

Explanation:

- `loadDictionary` reads `word.txt` and fills the `dictionary` 2D array. It returns the count of words loaded.
- If something went wrong (file missing, permission, or empty), so we print a red error (ANSI code) and exit with non-zero status and returns 1 cause the game cannot proceed without a dictionary of valid words.

3- Create pointer array for random selection

```c
// Create array of pointers to dictionary words
for (int i = 0; i < num_words; i++) {
    dict_ptrs[i] = dictionary[i];
}
```

Explanation:

- The 2D array `dictionary` is an array of rows; taking `dictionary[i]` gives a `char *` to that row.
- `pick_random_word` expects `char **` (array of `char*`), so we prepare `dict_ptrs` filled with pointers to each word we don't copy strings, we copy pointers.

Welcome header and legend

```c
/* Welcome (pink) and one-time legend */
printf("\033[1;35m=== Welcome to Wordle! ===\n\033[0m");
printf("You have %d attempts to guess the %d-letter word.\n\n", max_attempts, WORD_LEN);
printf("Legend (shown once): (\033[1;32m Green\033[0m = correct position, \033[1;33m Yellow\033[0m = wrong position,
```

Explanation:

- Uses ANSI color codes to print a colored welcome line (pink), informational line about attempts and word length, and a legend showing what colors mean.
- Green = correct letter correct place
- Yellow = correct letter wrong place
- Gray = not in the word
- This block is printed once at the start of the program to orient the player.

Board initialization and keyboard state

```c
/* persistent small board to show past guesses */
char past_guesses[6][WORD_BUFFER];
char past_feedbacks[6][WORD_BUFFER];
```

```
for (int i = 0; i < max_attempts; i++) {
    past_guesses[i][0] = '\0';
    past_feedbacks[i][0] = '\0';
}
int printed_legend = 1; /* we already printed it above */
/* keyboard state: 'U' unknown, 'G' green, 'Y' yellow, 'B' gray */
char keyboard_state[26];
for (int i = 0; i < 26; ++i) keyboard_state[i] = 'U';
```

Explanation:

```
char past_guesses[6][WORD_BUFFER];
char past_feedbacks[6][WORD_BUFFER];
for (int i = 0; i < max_attempts; i++) {
past_guesses[i][0] = '\0';
past_feedbacks[i][0] = '\0';
}
```

- `past_guesses` and `past_feedbacks` store up to 6 previous guesses and their feedback, enabling the board to be redrawn each loop so each attempt later displayed on a grid (like Wordle board) .
- Initialize each stored string to empty by setting `[0] = '\0'`.
- `printed_legend` flags that the legend was already shown (used when rendering UI).

  ```
  char keyboard_state[26];
  for (int i = 0; i < 26; ++i) keyboard_state[i] = 'U';
  ```

- `keyboard_state` tracks the best-known state for each letter:
  - 'U' = unknown (default)
  - 'G' = green (correct in place)
  - 'Y' = yellow (in word, wrong place)
  - 'B' = gray (not in word)

Pick random secret word

```
// Pick a random word
secret_word = pick_random_word(dict_ptrs, num_words);
if (secret_word == NULL) {
    printf("\033[1;91mError! Could not select a word.\n\033[0m");
    return 1;
}
```

Explanation:

- Calls `pick_random_word` to choose one random word from the dictionary and save it into "secret-word". which returns a `malloc`-allocated copy of a random dictionary entry.
- If allocation or selection fails (`NULL`), prints an error and exits.
- We'll `free(secret_word)` at the end of `main()` to avoid leaks.

Game loop

```
while (attempts < max_attempts && !won) {
/* Clear screen and re-render compact UI (legend shown only once) */
printf("\033[2J\033[H");
/* header */
printf("\033[1;35m=== Wordle ===\033[0m\n");
printf("Attempts: %d/%d\n", attempts, max_attempts);
if (printed_legend) {
/* show brief note that legend was shown once */
printf("(Legend shown at start)\n\n");
} else {
printf("\n");
}
```

Loop condition:

- `while (attempts < max_attempts && !won)` — run until player wins or uses all attempts.

  Inside the loop we:

1. Clear the screen and print header.
2. Render previous guesses and the small board.
3. Render keyboard (QWERTY layout) with colors.
4. Prompt and read player guess.
5. Validate guess.
6. Compute feedback and store it.
7. Update keyboard state.
8. Display feedback and check for win.

Clear screen and header

```
/* Clear screen and re-render compact UI (legend shown only once) */
printf("\033[2J\033[H");
/* header */
printf("\033[1;35m=== Wordle ===\033[0m\n");
printf("Attempts: %d/%d\n", attempts, max_attempts);
if (printed_legend) {
    printf("(Legend shown at start)\n\n");
} else {
    printf("\n");
}
```

Explanation:

- `"\033[2J\033[H"` — ANSI sequence to clear terminal and move cursor to top-left, giving a fresh UI each loop.

```
/* header */
printf("\033[1;35m=== Wordle ===\033[0m\n");
printf("Attempts: %d/%d\n", attempts, max_attempts);
if (printed_legend) {
```

- Header shows game title (magenta) and attempts used(chances)

```
/* show brief note that legend was shown once */
printf("(Legend shown at start)\n\n");
} else {
printf("\n");
}
```

- If legend was printed at program start we show a note instead of repeating it.

Render previous guesses board

```
/* Render board */
printf("Previous guesses:\n");
for (int i = 0; i < max_attempts; i++) {
    printf("%d: ", i + 1);
    if (past_guesses[i][0] == '\0') {
        for (int k = 0; k < WORD_LEN; k++) printf("[   ] ");
        printf("\n");
    } else {
        display_feedback(past_feedbacks[i], past_guesses[i]);
        printf("\n");
    }
}
```

Explanation:

- For each of the 6 rows we either print empty boxes `[ ]` or the previous guess with colored tiles using `display_feedback()`.
- This recreates a compact Wordle board in the terminal showing past attempts and their feedback.

Render keyboard (QWERTY)

```
/* Render keyboard (QWERTY) */
const char *row1 = "qwertyuiop";
const char *row2 = "asdfghjkl";
const char *row3 = "zxcvbnm";
printf("\nKeyboard:\n");
for (int r = 0; row1[r]; ++r) {
    char ch = row1[r];
    char s = keyboard_state[ch - 'a'];
    if (s == 'G') printf("\033[30;42m %c \033[0m ", ch);
    else if (s == 'Y') printf("\033[30;43m %c \033[0m ", ch);
    else if (s == 'B') printf("\033[37;100m %c \033[0m ", ch);
    else printf("[ %c ] ", ch);
}
```

```
printf("\n");
// row2, row3 printed similarly...
```

Explanation:

```
const char *row1 = "qwertyuiop";
const char *row2 = "asdfghjkl";
const char *row3 = "zxcvbnm";
printf("\nKeyboard:\n")
```

- The keyboard is printed in three rows (QWERTY). For each letter we look up its state in `keyboard_state`.

  ```
  for (int r = 0; row1[r]; ++r) {
  char ch = row1[r];
  char s = keyboard_state[ch - 'a'];
  if (s == 'G') printf("\033[30;42m %c \033[0m ", ch);
  else if (s == 'Y') printf("\033[30;43m %c \033[0m ", ch);
  else if (s == 'B') printf("\033[37;100m %c \033[0m ", ch);
  else printf("[ %c ] ", ch);
  }
  printf("\n");
  for (int r = 0; row2[r]; ++r) {
  char ch = row2[r];
  char s = keyboard_state[ch - 'a'];
  14
  if (s == 'G') printf("\033[30;42m %c \033[0m ", ch);
  else if (s == 'Y') printf("\033[30;43m %c \033[0m ", ch);
  else if (s == 'B') printf("\033[37;100m %c \033[0m ", ch);
  else printf("[ %c ] ", ch);
  }
  printf("\n");
  for (int r = 0; row3[r]; ++r) {
  char ch = row3[r];
  char s = keyboard_state[ch - 'a'];
  if (s == 'G') printf("\033[30;42m %c \033[0m ", ch);
  else if (s == 'Y') printf("\033[30;43m %c \033[0m ", ch);
  else if (s == 'B') printf("\033[37;100m %c \033[0m ", ch);
  else printf("[ %c ] ", ch);
  }
  printf("\n\n");
  printf("\nEnter your guess (attempt %d/%d): ", attempts + 1, max_attempts);
  ```

- Depending on the state we print colored blocks (ANSI codes) or plain brackets for unknown letters.
- This gives a quick view of which letters are confirmed, maybe, or eliminated.

Get user input

```
printf("\nEnter your guess (attempt %d/%d): ", attempts + 1, max_attempts);

// Get user input
if (fgets(guess, sizeof(guess), stdin) == NULL) {
```

```
    printf("\033[1;91mError reading input.\n\033[0m");
    continue;
}

// Remove newline character if present
if (guess[strlen(guess) - 1] == '\n') {
    guess[strlen(guess) - 1] = '\0';
}
```

Explanation:

- `fgets()` reads a line safely into `guess` (prevents buffer overflow).
- guess[strlen(guess) - 1] = ";removes `\n`.
- We trim the trailing newline by replacing it with '\0' so the guess string is clean for validation.
- If `fgets` returns `NULL`, we print an error and continue to next loop iteration.

Validate guess

```
if (!isValidGuess(guess, dictionary, num_words)) {
    printf("\033[1;91mError! Invalid guess. Must be a 5-letter word in the dictionary.\n\033[0m\n");
    continue;
}
```

Explanation:

- `isValidGuess` checks: correct length (`WORD_LEN`), alphabetic characters, and existence in the dictionary.
- If invalid, print a red error message and prompt the user again (no attempt penalty in this code as written).

Generate feedback and store

```
// Get feedback
getFeedback(secret_word, guess, feedback);

// Store and show in board
strncpy(past_guesses[attempts], guess, WORD_BUFFER - 1);
past_guesses[attempts][WORD_BUFFER - 1] = '\0';
strncpy(past_feedbacks[attempts], feedback, WORD_BUFFER - 1);
past_feedbacks[attempts][WORD_BUFFER - 1] = '\0';
```

Explanation:

- `getFeedback` fills `feedback` with 'G', 'Y', 'B' for each letter.
- We store the guess and feedback safely into the `past_guesses` and `past_feedbacks` arrays using `strncpy()` and ensure termination.
- This preserves the data to redraw the board each loop.

Update keyboard state (priority G ¿ Y ¿ B)

```
for (int i = 0; i < WORD_LEN; ++i) {
    char c = guess[i];
    if (c < 'a' || c > 'z') continue;
    int idx = c - 'a';
    char f = feedback[i];
    if (f == 'G') keyboard_state[idx] = 'G';
    else if (f == 'Y') {
        if (keyboard_state[idx] != 'G') keyboard_state[idx] = 'Y';
    } else { /* B */
        if (keyboard_state[idx] != 'G' && keyboard_state[idx] != 'Y') keyboard_state[idx] = 'B';
    }
}
```

Explanation:

- For each char in the guess, map it to an index (0..25).
- Update keyboard state with priority rules:
  - If currently Green, keep Green.
  - If Yellow and not Green, keep/update Yellow.
  - Gray only if letter hasn't been marked Green or Yellow.
- This prevents downgrading a letter that was previously confirmed better.

Display feedback and win check

```
printf("Feedback: ");
display_feedback(feedback, guess);
printf("\n\n");

// Check if the guess is correct
if (strcmp(guess, secret_word) == 0) {
    printf("\033[1;32m Congratulations! You guessed the word: %s\n\033[0m", secret_word);
    won = 1;
} else {
    attempts++;
}
```

Explanation:

- Uses `display_feedback` to print the colored tiles for current guess.
- If the guess equals the secret word (`strcmp==0`), print a green success message and set `won=1`.
- Otherwise increment the `attempts` counter and continue.

Game over and cleanup:

```
if (!won) {
    printf("\033[1;91m Game Over! The word was: %s\n\033[0m", secret_word);
}
```

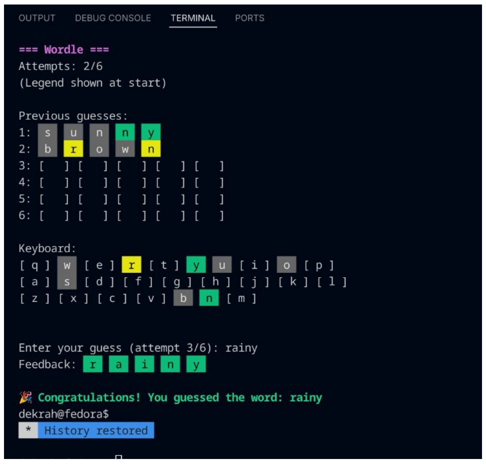
```
// Free allocated memory
free(secret_word);

return 0;
```

Explanation:

- If the player didn't win, reveal the secret in red.
- Free dynamically allocated `secret_word` to avoid memory leak.
- Return 0 → normal program termination.
- COMPLEXITY:

```
The \verb|main| function has a time complexity of \textbf{O(N × L)}
where N is the number of words in the dictionary and L is the word length
```

Program execution :



```
=== Wordle ===
Attempts: 2/6
(Legend shown at start)

Previous guesses:
1:  s   u   n   n   y
2:  b   r   o   w   n
3: [   ] [   ] [   ] [   ] [   ]
4: [   ] [   ] [   ] [   ] [   ]
5: [   ] [   ] [   ] [   ] [   ]
6: [   ] [   ] [   ] [   ] [   ]

Keyboard:
[ q ]  w  [ e ]  r  [ t ]  y   u  [ i ]  o  [ p ]
[ a ]  s  [ d ] [ f ] [ g ] [ h ] [ j ] [ k ] [ l ]
[ z ] [ x ] [ c ] [ v ]  b   n  [ m ]


Enter your guess (attempt 3/6): rainy
Feedback:  r   a   i   n   y

🎉 Congratulations! You guessed the word: rainy
dekrah@fedora$
 *  History restored
```

Figure 1: Final Wordle Program Running