Appendices

Appendix B: CS 108 Artifacts

B1: CS 108 – Course Schedule (from Fall 2025 Syllabus)

Week	Dates	Topic	Assignment Released	Assignment Due
1	8/25 - 8/31	Overview of Computers and Programming	8/25 – Lab 1 8/25 – Problem Set 01	
2	9/1* - 9/7	Overview of C	9/1 – Problem Set 02	9/3 – Lab 1 9/3 – Problem Set 01
3	9/8 - 9/14	Top-Down Design with Functions	9/8 – Problem Set 03	9/10 – Problem Set 02
4	9/15 - 9/21	9/19 - Exam 1 (Covers weeks 1 – 3)	Practice Exam 1 Released 9/12 Reviewed 9/17	9/17 – Problem Set 03
5	9/22 - 9/28	Selection Structures	9/22 – Lab 2 9/22 – Problem Set 04	
6	9/29 - 10/5	Repetition Structures	9/29 – Problem Set 05	10/1 – Problem Set 04 10/5 – Lab 2
7	10/6 - 10/12	Pointers and Modular Programming	10/6 – Midterm Project 10/6 – Problem Set 06	10/8 – Problem Set 05
8	10/13* - 10/19	10/17 - Exam 2 (Covers weeks 5 – 7)	Practice Exam 2 Released 10/10 Reviewed 10/15	10/15 – Problem Set 06
9	10/20 - 10/26	Array Pointers	10/20 – Lab 3 10/20 – Problem Set 07	10/26 – Midterm Project
10	10/27 - 11/2	Strings	10/27 – Problem Set 08 10/27 – Lab 4	10/29 – Problem Set 07 11/2 – Lab 3
11	11/3 - 11/9	Recursion		11/5 – Problem Set 08 11/9 – Lab 4
12	11/10 - 11/16	Structures and Personal Libraries	11/10 – Problem Set 09	
13	11/17 - 11/23	11/21 - Exam 3 (Covers weeks 9 – 12)	11/17 – Final Project Practice Exam 3 Released 11/14 Reviewed 11/19	11/19 – Problem Set 09
14	11/24* & 12/1 - 12/7	Dynamic Data Structures	12/1 – Problem Set 10	12/7 – Problem Set 10
15	12/8 - 12/11	Finals (No Classes) Please note there is not a final exam		12/8 @ 10:00am Final Project

* No Classes

9/1 → Labor Day;   10/13 → Mid-Semester Break;   11/25 & 11/27 → Thanksgiving Recess

<u>B2: Wordle-in-the-Terminal Final Project</u>

Note to reviewer: This excerpt is the instructions/ scoring rubric of the CS108 Final Project.

Final Project - Wordle

In this final project, you will create a terminal-based version of the popular game Wordle using the C programming language. You will use multiple files to structure your code, including a header file and implementation file. This assignment is designed to assess your understanding of C programming concepts covered throughout the course, including functions, arrays, structures, and file I/O.

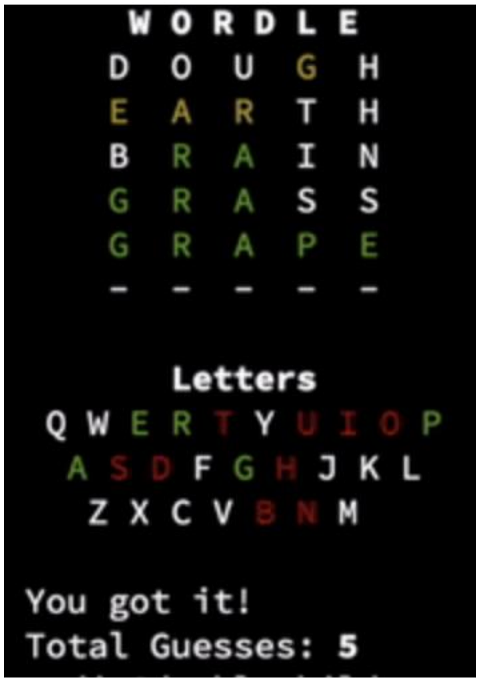
Assignment Overview

Let's talk about WORDLE! The game selects a secret 5-letter word, and the player has to guess the word within six attempts. After each guess, the game provides feedback by coloring the letters:

- **Green**: Correct letter in the correct position.
- **Yellow**: Correct letter in the wrong position.
- **Red**: Incorrect letter.

If you haven't played wordle I suggest giving it a go at this link. This will give you a better idea of the intended game mechanics.

A functioning final version of your game will look like this.



```
       W O R D L E
      D   O   U   G   H
      E   A   R   T   H
      B   R   A   I   N
      G   R   A   S   S
      G   R   A   P   E
      -   -   -   -   -

         Letters
   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

You got it!
Total Guesses: 5
```

Note to reviewer: This is an animated run through of a full game in the project environment.

Introduction

In this assignment, you will implement the core logic for a Wordle game that runs in a text-based terminal environment. You will be working with C, focusing on file I/O, string manipulation, arrays, structures, and modular programming by implementing functions defined in a header file.

An example program is located in the root directory `workspace` called `wordle_example`. You are encouraged to run this program to get an idea of the functionality you are implementing.

Getting Started

You have been provided with some starter code files and a data file:

- `wordle.c`: Contains the `main` function and the game loop. **Do not modify this file.**
- `wordle_functions.h`: The header file containing macros, structure definitions, and function prototypes. You will add the structure definition and interface comments here. **Do not modify the existing function prototypes.**
- `wordle_functions.c`: The implementation file where you will implement the functions declared in `wordle_functions.h`. **Do not modify the pre-implemented functions.**
- `words.txt`: Contains 3103 5-letter words (1 word per line), which serves as the official dictionary for the game.

Your tasks are to...

1. Define the `game_state_t` structure in the `wordle_functions.h` file according to the specifications below.
2. Add clear and informative **interface comments** (documentation comments explaining purpose, parameters, pre/post conditions) for **all** functions declared in the `wordle_functions.h` header file.
3. Add a **program-level comment** at the top of `wordle.c`, `wordle_functions.h`, and `wordle_functions.c` including your name, course, assignment details, and a brief description of the file's purpose.
4. Implement the required functions within `wordle_functions.c` as detailed in the Implementation Instructions section.

Implementation Instructions

1) Define the `game_state_t` data type structure

Define the `game_state_t` structure within the `wordle_functions.h` file. It should have the following member components:

Component	Description
guesses	2D Array of MAX_ATTEMPTS x (WORD_LENGTH + 1) to store all guesses made by the player.
current_guess	1D Array of size WORD_LENGTH + 1 to store the guess currently being processed.
guess_statuses	2D Array of MAX_ATTEMPTS x WORD_LENGTH to store the status (e.g., correct, present, absent) of each letter in each guess.
secret_word	1D Array of size WORD_LENGTH + 1 to store the secret word for the current game.
alphabet_state	1D Array of size 26 to track the state (e.g., used, correct, present, absent) of each letter of the alphabet across all guesses.
guess_count	Integer to store the number of guesses made so far (0 to MAX_ATTEMPTS).
game_status	Integer to store the current status of the game (e.g., 0 for active, 1 for lost, 2 for won).

Function Implementation Details

You are required to implement the following functions in `wordle_functions.c`. Implement them according to the prototypes provided in `wordle_functions.h` and the detailed instructions below. Standard C libraries like `stdio.h`, `stdlib.h`, `string.h`, `time.h`, and `ctype.h` are included in `wordle_functions.c`.

Important: Before you start writing implementations, make sure you have scrolled down in this guide and red the section titled **Testing with Function Drivers**.

2. `void get_word_list(char words[][WORD_LENGTH + 1], const char *filename);`

- **Purpose**: Read the list of valid words from a specified file into a 2D array.
- **Parameters**:
  - `char words[][WORD_LENGTH + 1]`: A 2D array allocated in the calling function, large enough to store the expected number of words.
  - `const char *filename`: A string containing the name of the file to read the words from (e.g., “words.txt”).

- **Instructions**:
  - Attempt to open the file specified by the `filename` parameter in read mode.
  - **Error Check**: Check if the `fopen` function returned `NULL`. If it did, the file could not be opened; print an error message to and return from the function immediately.
    - *Hint: In void functions you can use `return` without a return value to achieve this.*
  - Use a loop that continues as long as `fscanf` successfully reads a string from the file.
    - Copy the word read into the next available row of the `words` array.
    - Increment a counter to track how many words have been read and stored.
  - After the loop finishes close the file.

3. `void get_secret_word(char *secret_word, int list_length, char word_list[][WORD_LENGTH + 1]);`

- **Purpose**: Select a secret word randomly from the provided list of words.
- **Parameters**:
  - `char *secret_word`: Pointer to a character array where the selected secret word will be stored. Ensure this array has space for `WORD_LENGTH + 1` characters.
  - `int list_length`: The number of words actually present in the `word_list` array.
  - `char word_list[][WORD_LENGTH + 1]`: The 2D array containing all valid words loaded from the file.
- **Instructions**:
  - Seed the random number generator using the current time exactly once at the beginning of this function.
    - *Hint: Use `srand(time(NULL));`.*
  - Generate a random integer index between `0` and `list_length - 1` (inclusive).
    - *Hint: Use the `rand()` function and the modulo operator.*
  - Copy the word located at the random index from the `word_list` array into the `secret_word` array.

4. `game_state_t initialize_game_state(char secret_word[]);`

- **Purpose**: Initialize a new `game_state_t` structure for the start of a game.

- **Parameters**:
    - `char secret_word[]`: The secret word selected for this game session.
- **Instructions**:
    - Declare a local variable of type `game_state_t`.
    - Initialize the `guesses` array
        - Each row of the guess array represents a single guess by the user, so a row will hold a 5 letter word and the null character.
        - When the game starts each character should be a `-`
    - Initialize the `guess_statuses` array:
        - This 2d array is parrallel to the `guesses` array. It will be responsible for holding the status of each individual character of each guess. This will determine the coloring of that letter in final game (ie. green for a correctly placed letter in the secret word)
        - Set each position in the `guess_statuses` array to `-1`.
    - Copy the provided `secret_word` parameter into the `secret_word` member of your local game state variable`.
    - Initialize `guess_count` to `0`.
    - Initialize `game_status` to `0`
        - This indicates the game is active.
    - Initialize the `alphabet_state` array:
        - This member of the struct keeps track of the status of the 26 letters that appear below the guesses in the UI. The same rules apply here as in `guess_statuses`. The key difference is this is a single dimensional array with one position for each character in the alphabet.
        - Initialize each position in `alphabet_state` to `-1`.
    - Return the fully initialized local `game_state_t` variable.

5. **`int get_index_in_string(char string[], char target);`**

- **Purpose**: Find the first index of a target character within a given string (this is linear search).
- **Parameters**:
    - `char string[]`: The null-terminated string to search within.
    - `char target`: The character to search for.
- **Instructions**:
    - Loop through the `string` character by character.
    - If the `target` is found, return the index.
    - If the loop completes without finding the `target` character, return `-1`.

6. `int get_index_in_word_list(char word[], int list_length, char word_list[][WORD_LENGTH + 1]);`

- **Purpose**: Find the index of a specific word within a given list (array) of words.
- **Parameters**:
  - `char word[]`: The word to search for.
  - `int list_length`: The number of words actually present in the `word_list` array.
  - `char word_list[][WORD_LENGTH + 1]`: The 2D array containing the list of valid words.
- **Instructions**:
  - Loop through the `word_list` array.
    - Compare the input `word` with the word at the current index in the `word_list`.
    - *Hint: Use `strcmp`.*
  - If `word` is found in `word_list` return the current index immediately.
  - If the loop completes without finding an identical word, return `-1`.

7. `void get_next_guess(game_state_t *game_state, int list_length, char word_list[][WORD_LENGTH + 1]);`

- **Purpose**: Prompt the user to enter their next guess and validate it against the word list.
- **Parameters**:
  - `game_state_t *game_state`: Pointer to the current game state structure.
  - `int list_length`: The number of words in the `word_list`.
  - `char word_list[][WORD_LENGTH + 1]`: The array of valid words.
- **Instructions**:
  - Initialize a local string variable that will temporarily hold the users guess.
    - This will help us make sure we don't prematurely copy in a guess that is too long into our `game_state`, which would potentially corrupt data in adjacent memory cells!
  - Use a loop to repeatedly prompt the user until a valid guess is entered.
  - Your loop should:
    - Print a prompt asking the user to enter a guess (e.g., `"Enter a guess: "`).
    - Read the user's input string into the temporary local string variable you made earlier.
    - Determine the length if the guess

        - If the guess had a length of 5
            - Loop char by char over the guess, convert the char to uppercase then place it in `game_state->current_guess`.
        - If the guess length was not 5 or if the (now uppercase) `game_state->current_guess` does not exists in the `word_list`, your loop should repeat.
            - *Hint: you wrote a function that checks if a word is in a given list.*
    - Once a valid word (one found in the list) is entered, the loop terminates, and the valid guess remains stored in `game_state->current_guess`.

8. **`void update_state(game_state_t *game_state);`**
    - **Purpose**: Update the overall game state after a valid guess has been made and processed.
    - **Parameters**:
        - `game_state_t *game_state`: Pointer to the current game state structure.
    - **Instructions**:
        - Copy the *previously validated* `game_state->current_guess` into the `guesses` array at the current `guess_count` index.
        - Call `calculate_guess_status()` function, passing the `game_state` pointer and the current `game_state->guess_count` as the index of the guess to evaluate.
        - Check if the `game_state->current_guess` is identical to the `game_state->secret_word`.
            - If they match set `game_state->game_status` to `2` (this indicates a win).
        - Increment the `game_state->guess_count` by `1`.
        - Check if the game is lost:
            - This is the case if `game_state->guess_count` is now equal to `MAX_ATTEMPTS` AND the `game_state->game_status` is *not* `2`
            - If this is the case then set `game_state->game_status` to `1` (this indicates a loss).

9. **`void calculate_guess_status(game_state_t *game_state, int guess_index);`**
    - **Purpose**: Calculate the status (correct position, present but wrong position, or absent) for each letter of the guess at the specified `guess_index` and update the game state accordingly.

This is a tricky function, take your time and don't give up

- **Parameters**:
    - `game_state_t *game_state`: Pointer to the current game state structure.
    - `int guess_index`: The index (row) in `game_state->guesses` corresponding to the guess being evaluated.

- **Instructions**:
    - Declare a local integer array `secret_letter_counts[26]` and initialize all its elements to `0`. This will track the counts of each letter available in the secret word.
    - Declare a local integer array `guess_status[WORD_LENGTH]` and initialize its elements to `0`. This will temporarily store the status for the current guess.
    - **Count Secret Word Letters**: Loop through `game_state->secret_word`. For each character, find its corresponding index in the alphabet (e.g., 'A' is 0, 'B' is 1,
        - *Hint: ALPHABET is a constant macro you can use here*
        - Once found increment the count at that index in `secret_letter_counts`.
    - **First Pass (Handle Green Letters - Those in the correct Position)**:
        - Loop from over the characters in the current guess.
        - If `guess[i]` is equal to `game_state->secret_word[i]`:
            - Set `guess_status[i]` to `2` (this indicates it should be green).
            - Find the alphabet index for `guess[i]`.
            - Decrement the count for this letter in `secret_letter_counts`. This "uses up" the letter so it can't also be marked yellow later if multiple instances exist.
    - **Second Pass (Yellow Letters - Present but Wrong Position)**:
        - Loop from over the characters in the current guess.
        - Only proceed if `guess_status[i]` is *not* already `2` (green).
        - Find the alphabet index for `guess[i]`.
        - Check if the count for this letter in `secret_letter_counts` is greater than `0`.
            - If yes:
                - Set `guess_status[i]` to `1` (yellow).
                - For this letter, decrement `secret_letter_counts` (this "uses up" one instance of the letter, this will allow us to account for letters that are in the secret word multiple times).
            - If no, set `guess_status[i]` to `o` (red).
    - **Update Game State**:
        - Copy the computed statuses from the local `guess_status` array into `game_state->guess_statuses` at the correct location.
        - Update the overall `alphabet_state`: Loop through the `guess` character by character.

        - Find the alphabet index for the character.
        - If the newly computed status `guess_status[i]` is numerically greater than the currently stored status in `game_state->alphabet_state`, update `game_state->alphabet_state` at that location to `guess_status[i]`. (This ensures greens override yellows, and yellows override unused/grey).

Testing with Function Drivers

For this assignment, you will be provided with separate C files (`test_{function_name}.c`) called **test drivers**. Each driver is designed to test one of your functions in isolation.
Why use drivers?

- **Incremental Testing:** They allow you to test each function as you write it, rather than waiting until the entire program is assembled. This makes debugging much easier.
- **Verification:** They help you verify that your function logic meets the basic requirements before integrating it into the larger game.
- **Assessment: There is no autograder for this assignment.** This means you will need to use the test drivers to determine if your implementations are working correctly. Running and passing the driver tests is crucial for demonstrating correctness.

How to Compile and Run a Driver:

1. **Placement:** Ensure the test driver file (e.g., `test_get_word_list.c`), your `wordle_functions.c`, `wordle_functions.h`, and any necessary data files (like `test_words.txt` if the driver uses it) are all in the **same directory**. This is true by default, so don't move any files around inside the `code` directory.
2. **Compilation:** Open a terminal in the `code` directory. Use the `gcc` compiler to compile the driver along with your implementation file. Link them together to create an executable. Example:

```
gcc test_get_word_list.c wordle_functions.c -o test_get_word_list
```

(Replace `test_get_word_list` with the specific driver you are compiling).

3. **Execution:** Run the compiled executable from the terminal, this is identical to the process we have used to run your program executables all semester:

```
./test_get_word_list
```

(Replace `test_get_word_list` with the name of the executable you created).

4. **Analysis:** Observe the output of the driver. If tests fail, use the information to debug your function implementation in `wordle_functions.c`, then re-compile and re-run the driver.

Repeat this process for all provided test drivers.

Grade Evaluation

Your grade for this assignment will be based on the following criteria (out of 100 points):

1. **Code Structure, Comments, and Style (30%)**
   - **Program-Level Comments (10%):** A comment block at the top of `wordle.c`, `wordle_functions.h`, and `wordle_functions.c` including your name, course, assignment details, and a brief description of each file's purpose.
   - **Interface Comments (10%):** Clear, comprehensive documentation comments in `wordle_functions.h` for *all* function prototypes, explaining their purpose, parameters (including pre-conditions), and what they return or modify (post-conditions).
   - **Code Structure and Readability (10%):** Consistent indentation, meaningful variable names, appropriate use of whitespace, and clear logical structure within your C code.
2. **Functionality and Correctness (70%)**
   Your program should compile and run correctly. Your program will be compiled and run manually to check for proper implementation. Make sure that your program works as expected **in codio** prior to submission.

Running Locally on Your Own Machine

You are encouraged to download your project files (`wordle.c`, `wordle_functions.h`, `wordle_functions.c`, `words.txt`, and function drivers) and compile/run the complete game locally.

- **Screen Clearing:** The `clear_screen()` function uses `system("clear");` which works on macOS and Linux terminals. **If you are using Windows**, you may will need to modify the `clear_screen()` function's implementation in `wordle_functions.c` to use `system("cls");` instead.

Additional Notes/Reminders

- **Testing:** Test your complete program thoroughly after passing the individual function drivers. Play the game multiple times to check different scenarios (winning, losing, invalid inputs).

Good luck!

Appendix C: CS 295 Artifacts

C1: CS 295 Course Syllabus:

Note to reviewer: This is the relevant portion of the syllabus that was approved by the curriculum committee.

CS 295: Artificial Intelligence Applications
Fall 2026
Department of Computer Science
SUNY Polytechnic Institute

Class Times and Location:
Mondays, Wednesdays, and Fridays
9:20 AM to 10:30 AM
Kunsela Hall A135

Required Textbook
"Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow"
by Aurélien Géron
3rd Edition, O'Reilly
ISBN: 9781098125974

Instructor Information
Name, Email, Office, and Phone:
Derek Taylor
taylordm@sunypoly.edu
Kunsela C120
315-525-7229

Office Hours:
After class, by appointment, or:
Monday and Wednesday 1pm - 3pm
Tuesday 8am - 10am

Course Description
Applying Artificial Intelligence techniques across academic disciplines enables innovative solutions to complex, real-world problems. This course provides an application-based introduction to Artificial Intelligence. A portion of the course reviews the Python programming language and introduces essential data structures, including queues, trees,

and graphs. Through hands-on projects, students will explore core AI applications using popular libraries such as NumPy, Pandas, Scikit-learn, and Keras/TensorFlow. Topics include predictive analytics (classification and regression), neural networks, pre-trained networks for image recognition, neuroevolutionary learning with genetic algorithms to optimize network parameters, and reinforcement learning. The course emphasizes the practical implementation and evaluation of AI techniques, preparing students for further studies in machine learning.

Student Outcomes

By the conclusion of this course, students are expected to:

1. Implement and analyze fundamental data structures including queues, trees, and graphs, along with associated algorithms like BFS and DFS.
2. Utilize software libraries to manipulate and prepare data for AI contexts.
3. Apply and evaluate supervised learning techniques (classification and regression) to build predictive models.
4. Build, train, utilize and optimize various neural networks.
5. Understand the core concepts of reinforcement learning.
6. Develop and evaluate practical AI applications.

Course Schedule (subject to change)

Week 1: Python Programming (variables, control flow, functions, built-in data structures)
Week 2: Python Programming (built-in data structures continued, classes, NumPy basics)
Week 3: Linear Data Structures (stacks, queues, priority queues, intro to Pandas)
Week 4: Non-Linear Data Structures & Search (trees, graphs, BFS, DFS)
Week 5: Predictive Analytics & Classification Basics (ML workflow, Scikit-learn, KNN, train/test split)
Week 6: Advanced Classification (logistic regression, SVM, evaluation metrics, data preprocessing)
Week 7: Regression Techniques (linear regression, decision trees, evaluation metrics)
Week 8: Neural Networks Basics (MLP, Keras/TensorFlow, simple NN classifier)
Week 9: Neural Networks for Vision
Week 10: Neural Networks for Vision
Week 11: Neuroevolutionary Learning
Week 12: Neuroevolutionary Learning
Week 13: Reinforcement Learning with Neural Networks
Week 14: Reinforcement Learning with Neural Networks

Assignments and Grading

Coursework will fall into three weighted categories as follows:

Projects (60%) - Hands-on applications of course material.
Exams (30%) - Midterm and Final exam with equal weighting
Problem Sets (10%) - Summative assessment for each topic