LAB 02A README

COLLABORATION: I collaborated with Daniel Mendoza, in the class we did the whole project together from the beginning to the end, we also helped each other fix the bugs, run the code in the terminal, and go to the tutoring section together.

LAB SUMMARY: In this lab, we were tasked with implementing a simple Hangman game in C based on a set of rules and requirements provided in the lab manual. The important aspects of the lab included:

1. Rule Validation: We needed to implement a function (`gameRun`) that performed various checks on the input word, including its length, character validity, and whitespace rules. If the input violated any of these rules, the function returned an error code corresponding to the specific rule that was violated.

2. Game Logic: The lab's core involved developing the game logic for Hangman, where the player had to guess letters and reveal the hidden word. The game tracked the number of incorrect guesses, known as "balloons," and terminated either in victory if the word was revealed or in defeat if all balloons were popped.

3. User Input Handling: We needed to handle user input, converting lowercase letters to uppercase and checking for valid inputs. Invalid inputs resulted in appropriate error messages.

4. Test Harness: The lab included a test harness in `game_test.c` where we could test our `gameRun` function with different words and scenarios. We had to ensure that the game behaved correctly and handled various edge cases.

5. Header File and Modularity: The lab required us to create a header file (`Game.h`) that defined constants and the function prototype for `gameRun`. This header file helped separate the interface from the implementation.

In summary, the lab aimed to teach us how to implement a game while adhering to strict rules and requirements for word validation and gameplay. It emphasized the importance of modularity, header files, and error handling in C programming. Additionally, we had to ensure that the test harness effectively verified the game's functionality.

APPROACH: My general approach to the lab was to carefully read the manual first to understand the requirements, rules, and overall structure of the Hangman game that needed to be implemented in C. My first steps were as follows:

1. Understanding Requirements: I started by thoroughly reading the lab manual to understand the specific rules for allowable words and the gameplay, as well as the functions and header files I needed to create.

2. Creating Header File: I created the `Game.h` header file to define constants like `MAX_WORD_LENGTH` and `MAX_WRONG_ATTEMPTS` and to declare the function prototype for `gameRun`. This helped separate the interface from the implementation.

3. Implementing gameRun Function: I implemented the core logic of the `gameRun` function in `Game.c`. I followed the rules provided in the lab manual to check word validity, handle user input, and manage the game's progress.

4. Testing with Test Harness: I added test words to the `game_test.c` test harness to verify that the `gameRun` function worked correctly. This step was crucial for identifying any issues and ensuring the game followed the specified rules.

What worked well during the lab:
- Careful reading of the manual helped in understanding the requirements and rules.
- Breaking the implementation into logical steps, such as word validation, game logic, and user input handling, made it easier to develop the game.

What went wrong:
- At times, I encountered issues related to indexing, especially when converting characters to uppercase and comparing them. Careful attention to array indices and boundary conditions was essential.

If I were to approach this lab differently:
- I would consider adding more detailed comments and explanations in the code to make it easier for others (and myself) to understand the logic, especially in complex sections of the code.
- I would also consider using additional functions for certain aspects of the game, such as character validation and user input handling, to make the code more modular and easier to read.
- To improve testing, I would add more test cases, including edge cases, to ensure the game handles all scenarios correctly.

Overall, reading the manual thoroughly and breaking the implementation into manageable steps were key to completing the lab.

FINAL SUMMARY: The implementation of the lab ended successfully. I spent approximately 4-5 hours working on it, including understanding the requirements, coding, testing, and refining the code. Here are my thoughts on the lab:

What I Liked:
1. The lab provided a clear and structured problem statement, which made it easier to approach.
2. I appreciated the emphasis on modularity by requiring the creation of a separate header file.
3. The rules for allowable words and gameplay added a challenging and engaging aspect to the lab.

What I Disliked:
1. One potential drawback was the lack of detailed comments or explanations in the provided code template. More comments could have helped me understand complex sections of the code.
2. While the lab manual covered the basic requirements, a more extensive discussion or examples during class could have been beneficial for students who might be new to C programming.

Was it a Worthwhile Lab?
Yes, it was a worthwhile lab as it helped reinforce important programming concepts like string manipulation, conditional statements, loops, and modular code structure. It also introduced the idea of creating and using header files.

Suggestions for Improvement:
1. Providing more detailed comments in the code template could be a helpful addition to assist students in understanding the logic.
2. Expanding on class discussions related to C programming concepts and offering more examples could better prepare students for this lab.

Hardest Parts:
The hardest parts of the lab were handling the different rules for allowable words, particularly the whitespace rules, and ensuring that the game followed these rules correctly. Converting characters to uppercase and handling user input effectively was also challenging.

Points Distribution:
The points distribution for grading seemed appropriate as it emphasized the key aspects of the lab, such as implementing the core game logic, word validation, and testing with the provided test harness.

In summary, the lab provided a valuable learning experience, reinforcing C programming skills and code modularity. With some minor enhancements like more extensive comments and additional in-class discussions on relevant concepts, it could become an even more effective learning opportunity. Overall, the lab was a good exercise in programming and problem-solving.

LAB SPECIFIC QUESTION: To modify string testing functions to gracefully handle cases where the string might not be null-terminated, you can incorporate explicit length parameters in the function signatures and ensure that the functions do not rely on null-termination. Here's a general approach to modifying string testing functions:

1. Update Function Signatures: Modify the function signatures to include a parameter for the length of the input strings. This way, you explicitly specify the length of the strings, and the functions won't rely on null-termination.

2. Use Length Information: Instead of relying on null-termination to determine the end of the string, use the provided length parameter to iterate through the characters in the string.

3. Check Length Constraints: Ensure that any string manipulation or comparison operations do not exceed the specified length. This is particularly important when dealing with functions that could potentially modify the string.

4. Return Appropriate Values: Adjust the return values or error codes to indicate when a string length is exceeded or if any other constraint is violated.

Here's an example of how you can modify a basic string testing function, such as one that checks for a valid character:

Original Function:
```c
int isCharacterValid(char c) {
    if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')) {
        return 1;  // Valid character
    }
    return 0;  // Invalid character
}
```

Modified Function to Handle Non-Null-Terminated Strings:
```c
int isCharacterValid(const char* str, int length, int index) {
    if (index >= 0 && index < length) {
        char c = str[index];
        if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')) {
            return 1;  // Valid character
        }
```

```
    }
    return 0;  // Invalid character or out of bounds
}
```

In this modified function, we pass the string `str`, its length `length`, and the index `index` as parameters. We explicitly check the bounds using the provided length information and do not rely on null-termination. This approach ensures that the function gracefully handles cases where the string might not be null-terminated and allows you to perform safe character checks and string manipulations.