Lab10

1. Lab Topics

This lab primarily covers multiple files and static member variables inside functions.

2. Static Member Variables – Bug Smasher Update.

Remember the Bug Smasher update in Lab09, for this lab, you are to further update the Bug smasher with the following details; you'll majorly be working with the getUserGuess() function.

1. Put a constraint on the number of guesses/attempts the player can make. That is, the player cannot make more than 10 attempts. **USE STATIC MEMBER VARIABLES** to keep track of the number of times the get user's guess function (getUserGuess()) is called and for each attempt, print out to the user the current number of attempts they've made. Once the number of attempts (using static member variables) get to 10, report to the user they have failed to kill the bug. Then ask the user if they want to play again.
2. In the getUserGuess() function, check the range of user input as you always have, but now instead of using an input validation loop to ask for user's input, **RECURSIVELY,** call the function getUserGuess().
3. Make sure to integrate your new updates with your previous code.

Example output 1:

```
Bug position1: 7
Bug position2: 6
Welcome to Bug Smasher! You have 10 attempts to guess the bug
positions.
Enter your guess (1-10) (Attempt 1): 1
The bug is to the left of your guess. Try a lower number.
Enter your guess (1-10) (Attempt 2): 2
The bug is to the left of your guess. Try a lower number.
Enter your guess (1-10) (Attempt 3): 3
The bug is to the left of your guess. Try a lower number.
The bugs have swapped positions! Bug 2 is laughing at you now.
Enter your guess (1-10) (Attempt 4): 4
The bug is to the left of your guess. Try a lower number.
Enter your guess (1-10) (Attempt 5): 3
The bug is to the left of your guess. Try a lower number.
Enter your guess (1-10) (Attempt 6): 2
The bug is to the left of your guess. Try a lower number.
The bugs have swapped positions! Bug 1 is laughing at you now.
Enter your guess (1-10) (Attempt 7): 1
The bug is to the left of your guess. Try a lower number.
Enter your guess (1-10) (Attempt 8): 1
The bug is to the left of your guess. Try a lower number.
Enter your guess (1-10) (Attempt 9): 2
The bug is to the left of your guess. Try a lower number.
The bugs have swapped positions! Bug 2 is laughing at you now.
You've used all your attempts. The bugs are still laughing at
you!
```

3. Multiple File

Take a look at how fast your code is growing (some having upto 500 lines of code), as the code grows, maintaining clarity, debugging issues, and collaborating with others become increasingly difficult. By adopting the practice of using multiple files, you can break your code into manageable, modular components, making it easier to understand, maintain, and collaborate on. This approach also enhances code reusability and promotes good coding practices, setting a strong foundation for their software development careers.

Task 1: Your task here is to break the huge monolithic code you have into multiple files, using the following as discussed in class. Remember to use the `#ifndef, #define, … #endif` directives.

1. Header file (extension .h)
    a. Function prototypes and documentations
    b. Global constants
    c. Templates code
2. Main file (extension .cpp)
3. Implementation file (extension .cpp)

Task 2: Generate any error in any function of the implementation file. Take a screenshot of the error and follow the process of debugging it.

Task 3: Make sure to compile your code and produce the output which should be exactly same as what you have with the monolithic code.

Compilation of multiple files from class note.

- g++ –Wall greetings.cpp number.cpp –o name_of_executable
- At compile time, compile all C++ files (wild card)
    - g++ *.cpp –o name_of_executable

NOTE: If you have not been writing function prototypes and their documentations, make sure to start writing it in this lab, else part of your grades will be deducted (-20).

General Hints:

1. Remember to initialize your variables appropriately and at the right location.
2. Implement input validation to ensure user input is within valid ranges.
3. Keep the code organized and use clear variable names.
4. Make sure the program gracefully exits when the user chooses to do so.
5. Feel free to copy the codes from your previous lab.

4. How to get full marks

To get a 100% on this lab your code should:

1. **Use static variables where instructed!**
2. Use good variable names such that one can easily understand a variable's purpose just by looking at the name.
3. The program needs to be intuitive (e.g., display proper messages while you are taking user input or printing the result)
4. Follow all good coding conventions such as proper indentation.
5. Adhere to all coding standards outlined in lab2.
6. Follow the instructions of cloning, making dir, and submitting your code to git as previously discussed in lab01 and lab02.
7. Comment your code properly (do not write comments for things that are obvious)
8. Push your most recent code in git and submit through canvas as well. The canvas submission should include following files:
    a. **The cpp file downloaded from git.**
    b. **At least 2 Image files**
        i. **Screenshot of the right result for the bug update following the example I have in example output 1.**
        ii. **Screenshot of the error generated.**