**Introduction**

To solve the solution, I chose to use the functional paradigm in GO, the imperative paradigm in Python and the procedural paradigm in C. The functional paradigm uses functions as building blocks for the program. It takes in parameters and returns a value. The functions are pure, which means that their outputs are solely determined by the input without modifying the external state of the program. The procedural paradigm revolves around procedures, these are set of instructions that carry out specific tasks. The procedures hold code that is reusable so that it is not repeated, and the code is efficient. I used pointers as parameters to store the output of the procedure to then be used by the program. The imperative paradigm works through the code from start to finish as step-by-step instructions.

**Comparing Procedural and Functional Paradigm**

Exploring the intricacies of functional and procedural paradigms reveals nuanced differences and similarities that significantly influence code structure and execution. In both paradigms, subroutines play a pivotal role by encapsulating reusable code. However, the functional paradigm distinguishes itself by ensuring that subroutines, or functions, return values, contributing to the paradigm's focus on immutability. Conversely, procedural subroutines do not return values but excel in organizing code for reuse.

Both functions and procedures, in the context of these paradigms, serve as modular entities encapsulating sets of instructions. For instance, a subroutine designed to choose a random word from a list was implemented in both paradigms. In the functional paradigm, the function version returned the word, maintaining purity and predictable outputs. In contrast, the procedural version stored the word in the address of one of the parameters passed, adhering to the paradigm's emphasis on reusable and efficient code.

A notable commonality between functional and procedural paradigms is their acceptance of parameters and their ability to be called from anywhere within the code. This flexibility enhances the modularity and adaptability of programs developed under these paradigms.

**Comparing Imperative and Procedural Paradigm**

Reviewing the relationship between procedural and imperative programming, it's evident that procedural programming serves as a specialised subset within the broader imperative paradigm. Both paradigms share the foundational principle of step-by-step execution, where the program unfolds in a linear process. However, procedural programming introduces a nuanced approach by emphasising the use of procedures, which are modular units of code that encapsulate specific tasks, to streamline and organise repetitive code segments.

In practical terms, during the implementation of the hangman game in C, where procedural programming was employed. The "displayWord" procedure grouped together the code responsible for showcasing the hangman word. This procedure, marked by its reusability, was called multiple times throughout the program, showcasing the procedural paradigm's emphasis on efficient code organisation and the encapsulation of repetitive tasks.

In contrast, the broader imperative paradigm, used in Python, executes instructions sequentially from start to finish. The flow of the program is primarily dictated by conditional statements, "if" selection statements, and iterative constructs like "while" and "for" loops. While both procedural and imperative paradigms share the commonality of step execution, the imperative paradigm prioritises the execution of the program's instructions rather than efficiency. For example, within my imperative code, I copied and pasted the code I created for the easy level into the other remaining levels and changed the necessary values.

**Comparing Functional and Imperative Paradigm**

Functional and imperative programming are significantly different. The functional paradigm places an emphasis on immutability, this is shown in my GO solution when appending a new guess letter onto the guessesLetters array slice rather than modifying the array itself. Immutability dictates that functions produce outputs solely determined by their input parameters. In contrast, the imperative paradigm, as seen in my Python implementation, embraces mutability, allowing variables to undergo dynamic changes during program execution. This mutable nature imparts a dynamic quality to the program, as variables can be modified and adapted to evolving circumstances. This is shown within the guessedLetters array in the python code, the new letter is just added onto the array.

The functional paradigm is more efficient in coding as the functions group together similar codes and can be called within each other. I wrote a playHangman function that is called with the parameters of the level from the main body of the program, and this then calls all the other function within. This makes the code more organised and efficient as repeated code isn’t used unlike imperative which has multiple reused codes throughout the program.

**Conclusion**

After completing the three solutions, it is obvious that each paradigm has its own advantages and disadvantages. Imperative is straightforward and easy to understand but it isn’t efficient and led to confusion from the amount of code produced from creating the hangman game as the levels had the same code but different variable values. Procedural is a better version of imperative, but it required extra variable to retrieve the results from the procedure. For example, when I needed to check if the word was correct, I had to pass in a result variable to get the result. Overall functional seemed the most better suited of all three as it was efficient and easy to understand, it did not require any extra variable as the function returns a value.