Lab 2: First Steps in Implementing the project (Made by: Amin Avan)

1. **Create 'myshell.c' file.** (This is the main source file of your shell application. It contains the core logic for interacting with the user, including reading commands from the user, interpreting these commands, and then invoking the appropriate actions or functions based on those commands. It's where the main function resides, which is the entry point of your program.)
    a. Step 1: Including the Header File (Includes the header file myshell.h which contains declarations for the functions and possibly global variables, constants, and other includes needed across your shell application. This allows myshell.c to use the functions defined in utility.c and elsewhere without directly including their definitions.)

    ```
    #include "myshell.h"
    ```

    b. Step 2: Defining the execute_command Function (Defines a function named execute_command that takes a single argument, input, which is a pointer to a character array (string). This function is responsible for parsing the input command and executing it accordingly.)

    ```
            void execute_command(char *input) {
            ...
            }
    ```

    i. In execute_command() – Tokenizing the Input (The input string is tokenized using the strtok function. delim contains the delimiters (space and newline) used to split the input into tokens. This is crucial for parsing commands and their arguments.)

    ```
            char *token;
            char *delim = " \n";
            token = strtok(input, delim);
    ```

    ii. In execute_command() – Processing the commands (The first token is checked to see if it's a recognized command (cd). If it is, the function attempts to get the next token, which represents the directory path, and calls change_directory to change the current working directory. If the command isn't recognized, it prints an error message.)

    ```
    if (token != NULL) {
      if (strcmp(token, "cd") == 0) {
        token = strtok(NULL, delim); // Get next part
    (directory path)
        change_directory(token);
      } else {
        printf("Command not recognized.\n");
      }
    }
    ```

    c. Step 3: The main Function (Defines the entry point of the program. It continuously prompts the user for input and processes it using execute_command. The loop continues until an EOF (End-of-File) is encountered, at which point the program exits.)
        i. Prompting for User Input (The printf statement is used to display a prompt (MyShell>) to the user. fgets is then used to read a line of input from the user into the input buffer. If fgets returns NULL, indicating EOF, the loop is exited, and the program ends.)
        ii. Processing the Input (Each line of input received from the user is passed to the execute_command function for processing.)

```
int main() {
  char input[1024]; // Buffer for user input

  while (1) {
    printf("MyShell> ");
    if (fgets(input, sizeof(input), stdin) == NULL) {
      break; // Exit on EOF
    }
    execute_command(input); // Process the input
  }

      return 0;
}
```

2. **Create 'utility.c' file** (This source file contains the implementations of utility functions declared in myshell.h. Functions like change_directory(char *path) would be defined here. These utility functions perform specific tasks that are called from the main program or elsewhere. Separating these functions into their own source file helps keep your code organized and modular.)
    a. Step 1: Including Necessary Headers (Each header file serves a specific role.
    myshell.h: Includes the function prototype for change_directory and potentially other shared functions and constants.
    stdio.h: Used for input and output functions, like printf and perror.
    stdlib.h: Might be included for other utility functions, although it's not directly used by change_directory.
    string.h: Included for string operations, though change_directory doesn't explicitly require it in the provided code snippet.
    unistd.h: Provides access to the POSIX operating system API, including the chdir and getcwd functions used in change_directory.)

    ```
    #include "myshell.h"
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <unistd.h>
    ```

    b. Step 2: Implementing the change_directory Function (Defines a function to change the shell's current working directory to path. If path is NULL, the function prints the current working directory instead.)

    ```
    void change_directory(char *path) {
    ...
    }
    ```

        i. In change_directory() – Handling a NULL Path Argument (When no specific directory is provided (i.e., path is NULL), the function fetches and prints the current working directory using getcwd. If getcwd fails, an error message is printed using perror.)

        ```
        if (path == NULL) {
        ```

```
    char cwd[1024];
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        printf("%s\n", cwd);
    } else {
        perror("getcwd() error");
    }
}
```

ii. In change_directory() – Changing to a Specified Directory (If a path is provided, chdir attempts to change the current working directory to path. On failure (returning non-zero), perror prints an error message, prefixed with "myshell", indicating the error source.)

```
else {
    if (chdir(path) != 0) {
        perror("myshell");
    }
}
```

3. **Create 'myshell.h' file** (This header file acts as an interface between different parts of your shell application. It typically contains declarations of functions, constants, and sometimes type definitions that are used across multiple source files. By including myshell.h in both myshell.c and utility.c, you ensure that both source files have access to the same function prototypes and definitions, facilitating consistency and linkage between different parts of your program.)

   a. Step 1: Preprocessor Guard (These lines are known as an "include guard." They prevent the header file from being included multiple times in the same file or across multiple files during compilation. If MYSHELL_H is not defined, it gets defined, and the contents of the header file are processed. This mechanism avoids potential issues like double declaration errors.)

```
#ifndef MYSHELL_H
#define MYSHELL_H
```

   b. Step 2: Standard Library Inclusions

```
#include <stdio.h>   // Standard I/O functions
#include <stdlib.h>  // Standard library for functions like malloc
#include <string.h>  // String handling functions
#include <unistd.h>  // Provides access to the POSIX operating
system API
```

   c. Step 3: Function Declarations (This line declares the prototype for the change_directory function. Declaring the function in the header file allows it to be visible and accessible across different parts of your shell program, assuming they include myshell.h. The comment clarifies that this function implements the functionality for the cd command within your shell.)

```
void change_directory(char *path); // declaration of 'cd' function
(cd function prototype)
```

   d. Step 4: End of Include Guard (This marks the end of the conditional inclusion started at the beginning of the file. It effectively closes the preprocessor directive that checks if MYSHELL_H was previously defined, ensuring the content of the header is only included once per compilation unit.)

```
#endif
```

4. **Create 'Makefile' file.** (A Makefile is used to automate the compilation and build process of your project. It defines a set of rules for building the executable from your source files. These rules include how to compile each source file, how to link them together, and any other commands needed to build the application. By running a command like make in the terminal (in the directory where the Makefile is located), the make utility reads this Makefile to compile the source code into an executable program according to the instructions defined within it. This simplifies the build process, especially as your project grows in complexity.)
   a. Step 1: Defining the Default Target (This line defines the default target that make will build if no specific target is specified on the command line. The target all depends on the myshell target, meaning that if you just run make without arguments, it will attempt to build myshell.)

```
all: myshell
```

   b. Step 2: Compiling the Shell Program

```
myshell: myshell.c utility.c
	gcc -o myshell myshell.c utility.c -I.
```

5. **Create 'readme' file.**
   a. The readme file (often named README.md if it uses Markdown formatting) provides documentation for your project. It typically includes information such as an overview of the project, installation instructions, usage examples, features, and any other important information that users or developers might need to know to understand or use your shell. The README is the first thing people will see when they look at your project repository, so it's an important tool for communication and documentation.
6. **Compiling Your Shell.**
   a. Open a Linux-terminal and navigate to the directory containing your shell's source codes and 'Makefile' file.
   b. Type 'make' and press Enter. (The Makefile you've provided compiles myshell.c and utility.c into an executable named myshell. If there are any compilation errors, they will be displayed in the terminal. You'll need to fix these errors before proceeding.)
   c. After successful compilation, you can start your shell by typing ./myshell in the terminal and pressing Enter.
   d. Since you've implemented the cd command, try changing directories to test its functionality. For example, typing cd [valid folder name] followed by Enter should change the current working directory to [valid folder name].