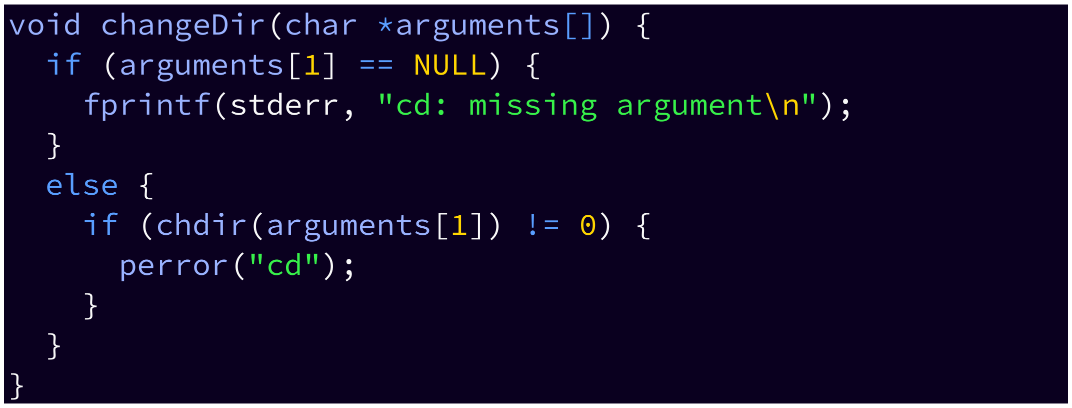
This is an implementation of a custom shell in c. The shell follows instructions set and assigned in Codio to execute commands from the command line

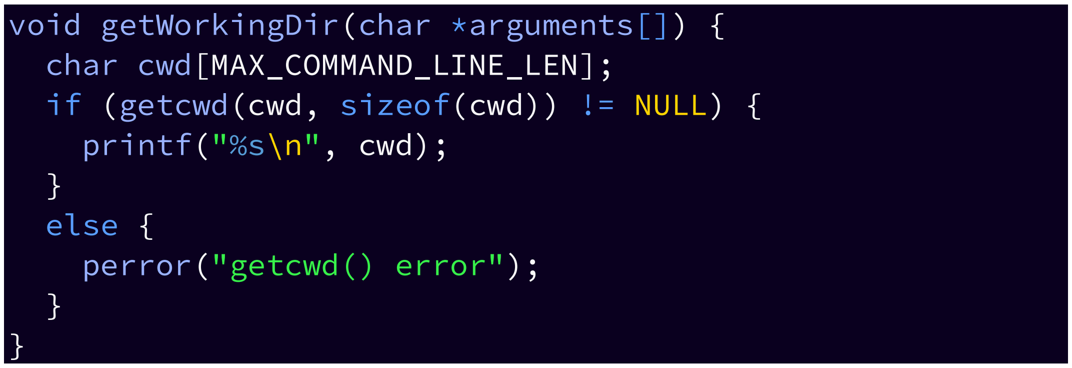
Task 1 : Printing the Shell Prompt and Adding Built-in Commands

1. cd: changes the current working directory
    a. changeDir is a void function with a char pointer to the array of command line arguments. It uses the inbuilt chdir function to change the current working directory to the specified location if it exists.
    b. Returns an error message if directory location doesn't exist.
    c. Implementation:



```
void changeDir(char *arguments[]) {
  if (arguments[1] == NULL) {
    fprintf(stderr, "cd: missing argument\n");
  }
  else {
    if (chdir(arguments[1]) != 0) {
      perror("cd");
    }
  }
}
```

2. pwd: prints the current working directory
    a. getWorkingDir is a void function with a char pointer array of command line arguments. It uses the inbuilt getcwd function to print the current working directory.
    b. Implementation:



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

3. env: prints the current values of the environment variables
    a. getEnv is a void function with a char pointer array of command line arguments.
    b. In the command line:

    i. If there is an argument after ‘env’ in the command line, the inbuilt function getenv is used to retrieve the value of the argument from the environment variables.
    ii. Else, all the environment variables are printed to the command line.
  c. Implementation:

```
void getEnv(char *arguments[]) {
  if (arguments[1] != NULL) {
    printf("%s\n", getenv(arguments[1]));
  }
  else {
    int i = 0;
    while(environ[i]) {
      printf("%s\n", environ[i++]); // prints in form of
"variable=value"
    }
  }
}
```

4. exit: terminates the shell.
  a. exitShell is a void function with a char pointer array of command line arguments. It used the inbuilt exit function.
  b. Implementation:

```
void exitShell(char *arguments[]) {
  exit(0);
}
```

5. setenv: sets an environment variable.
  a. setEnv is a void function with a char pointer array of command line arguments. It uses the inbuilt setenv function to set environment variables and edit them if they exist. The characters after “=” in the arguments after setenv in the command line is set as the value to the characters before “=” (the key).
  b. Returns an error message if inbuilt setenv fails.
  c. Implementation:

```
void setEnv(char *arguments[]) {
  if (arguments[1] == NULL) {
    fprintf(stderr, "setenv: missing argument\n");
  }
  else {
    char *list[2];
    int i = 0;

    char *ptr = strtok(arguments[1], "=");

    while(ptr != NULL) {
      list[i] = ptr;
      i++;
      ptr = strtok(NULL, "=");
    }

    if (setenv(list[0], list[1], 1) != 0) {
      perror("setenv");
    }
  }
}
```

6. echo: prints message and values of environment variables to the command line.
   a. echoFunc is a void function with a char pointer array of command line arguments.
   b. There are two conditions for this command:
      i. If the variable is defined and stored as an environment variable, on the command line, the variable is started with “$” and echo prints the value of the environment variable to the command line. The inbuilt function getenv is used to get the values for the environment variables.
      ii. Otherwise, the arguments after echo are printed to the command line.
   c. Returns an error message if getenv fails.
   d. Implementation:

```
void echoFunc(char *arguments[]) {
  if (arguments[1] == NULL) {
    fprintf(stderr, "echo: missing argument\n");
  }
  else {
    int x = 1;

    while (arguments[x] != NULL) {
      if (arguments[x][0] == '$') {
        printf("%s ", getenv(arguments[x]+1));
      }
      else {
        printf("%s ", arguments[x]);
      }
      x++;
    }
    printf("\n");
  }
}
```

Task 2 : Adding Processes

The following lines of code create a fork to add a process. The function cmd_exec is used to run the built-in commands implemented above.
Implementation:

```
// Fork a child process to execute the command.
      int pid;
      if ((pid = fork()) == 0) {
        // Child process.
        signal(SIGINT, SIG_DFL);
        cmd_exec(arguments);
        exit(0);
      }
```

Task 3 : Adding Background Processes

The boolean variable ‘background’ is used to check for the presence of a background process. If there is ‘&’ as the last argument in the command line, background is set to true, otherwise false.

Implementation:

```
char *last_token = arguments[position-1];
    bool background = false;
    if (strcmp(last_token, "&") == 0) {
      background = true;
      arguments[position-1] = NULL;
    }
```

Task 4 : Signal Handling

The void function signal_handler was created to handel the signal call for Ctrl + C. This function takes in an integer representing the signal number. It terminates a foreground process. It is useful to prevent the shell from exiting unexpectedly.

Implementation:

```
void signal_handler(int signum) {
  if (cmd_pid != -1) {
    kill(cmd_pid, SIGINT);
  }
}
```

Task 5 : Killing off Long Running Processes

timeout_process is a void function with parameters time and pid (both integers). This function is called to terminate the foreground process after 10 seconds of process incomletion.

Implementation:

```
void timeout_process(int time, int pid) {
  sleep(time);
  printf("Foreground process timed out\n");
  kill(pid, SIGINT);
}
```