CPS1012 – Operating Systems and Systems Programming 1 – Assignment Report

Edward Sciberras

Introduction

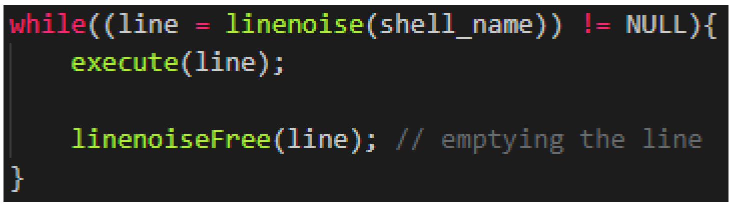
This project was a product of the CPS1012 module of Operating Systems and Systems Programming. The aim of this project was to create a small shell that is capable of internal commands that are pre-programmed and if the command is not part of the internal library, then try execute that command as an external command in the UNIX shell that is above it.

The shell is also capable of piping commands that are external as well as redirecting outputs to files that will write to or append to said text file. Text files could also be used as an input for commands.

Input, Tokenisation and Parsing

The scope of this part of the algorithm is to prompt the user and showing them that the program is ready to take in inputs from them. Once the user will input their desired text the algorithm will then have to process the text and alter it in such a way that the program will be able to use it to further complete other tasks.

The input of the user is saved using the 'linenoise' library that was provided to us. Linenoise will get the line and save it as a string. The algorithm will keep on prompting the user to input a command until they type in the exit keyword which will exit the whole program.

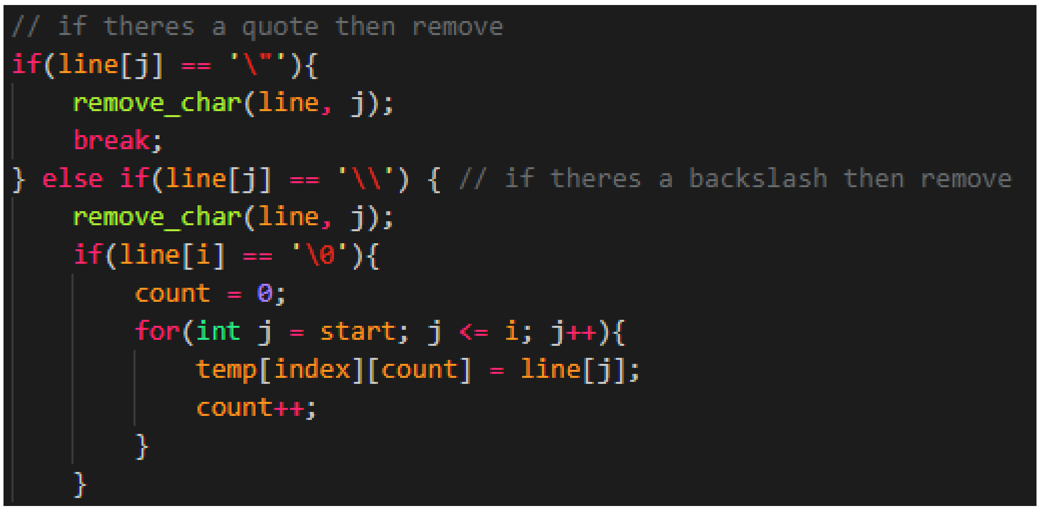


```
while((line = linenoise(shell_name)) != NULL){
    execute(line);

    linenoiseFree(line); // emptying the line
}
```

Figure 1 - Getting the users input every time in a loop.

An important part of 'cleaning' the input is tokenisation, which is essentially splitting the words as separate entities by certain characters that are hard coded into the program. Essentially the line in string form will be passed as a parameter and whenever one of the delimiters if found, it will create a singular string from all the characters before the delimiter and start again after the delimiter. Delimiters would include pipe symbols, colons, spaces, tabs and angled brackets.

Quoting is also an important part of parsing and tokenising as there are special situations in which some variables or words need to be printed as the literal value or as the value in the variable. For instance, when there is a character after a backslash it will remove the backslash and print the character after only. Enclosing characters with quotes will also print the literal values of the characters inside.



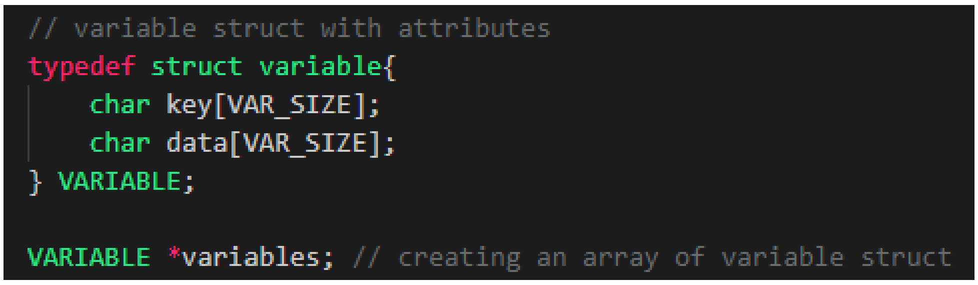
```c
// if theres a quote then remove
if(line[j] == '\"'){
    remove_char(line, j);
    break;
} else if(line[j] == '\\') { // if theres a backslash then remove
    remove_char(line, j);
    if(line[i] == '\0'){
        count = 0;
        for(int j = start; j <= i; j++){
            temp[index][count] = line[j];
            count++;
        }
    }
```

Figure 2 - Removing the backslashes and quotes from text when parsing.

Parsing and printing or variables is also important, so that when a dollar sign is encountered the parsing will not operate as usual and will have a special set of rules as opposed to the normal rules.

Variables

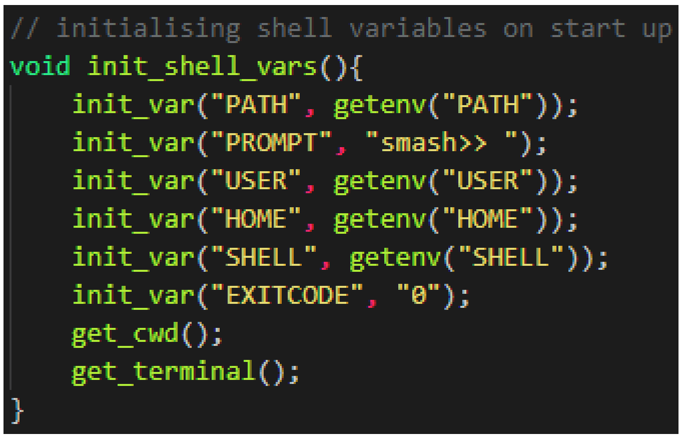
Each variable is made from a struct that is in the header file for the variables file. An array of struct is then made in order to keep track of all of them. The struct has two attributes; they key and the data. The key part is how each variable will differ as no two variables can have the same name as each other. The data part is what the variable will store.



```c
// variable struct with attributes
typedef struct variable{
    char key[VAR_SIZE];
    char data[VAR_SIZE];
} VARIABLE;

VARIABLE *variables; // creating an array of variable struct
```

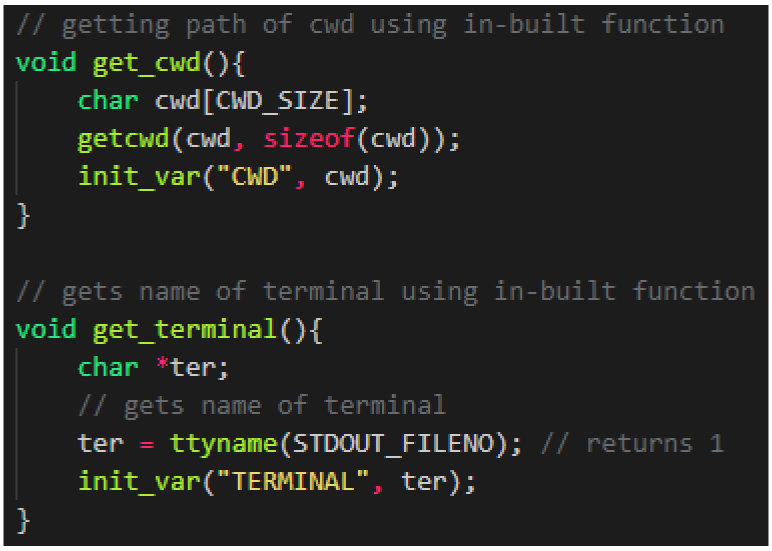
Figure 3 - Variable struct and array of variables.

Shell variables are special variables that are initiated on start-up of the shell, these hold a special purpose to the environment. The shell variables are the; PATH, PROMPT, CWD, USER, HOME, SHELL, TERMINAL, EXITCODE.



```
// initialising shell variables on start up
void init_shell_vars(){
    init_var("PATH", getenv("PATH"));
    init_var("PROMPT", "smash>> ");
    init_var("USER", getenv("USER"));
    init_var("HOME", getenv("HOME"));
    init_var("SHELL", getenv("SHELL"));
    init_var("EXITCODE", "0");
    get_cwd();
    get_terminal();
}
```

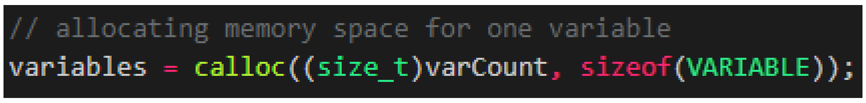
Figure 5 - Initiating shell variables on start up.



```
// getting path of cwd using in-built function
void get_cwd(){
    char cwd[CWD_SIZE];
    getcwd(cwd, sizeof(cwd));
    init_var("CWD", cwd);
}

// gets name of terminal using in-built function
void get_terminal(){
    char *ter;
    // gets name of terminal
    ter = ttyname(STDOUT_FILENO); // returns 1
    init_var("TERMINAL", ter);
}
```

Figure 4 - Methods to get CWD and the terminal.

The user can input a variable by following the scheme of “name=value”. There can not be any white space between the equals and the words. When a new variable is made, the ‘calloc’ function is called to dynamically allocate more memory for each new variable, this makes the number of variables more efficient and better use of memory as opposed to having a fixed amount of memory at all times. The algorithm will also check if the variable with they same key exists and if so, it will overwrite the value.

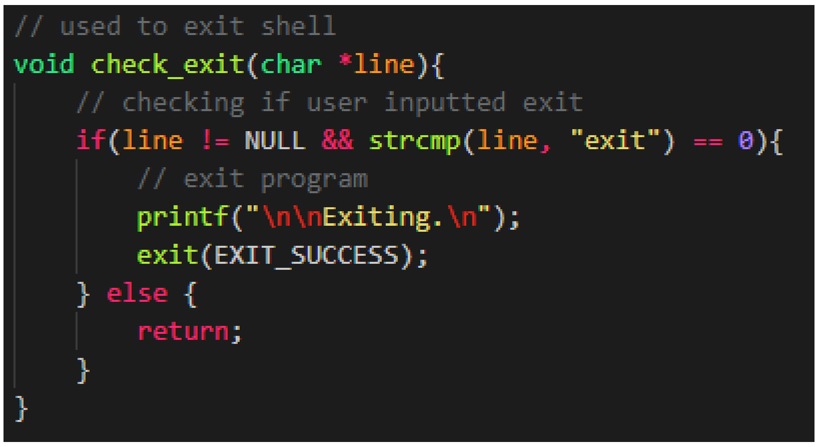


```
// allocating memory space for one variable
variables = calloc((size_t)varCount, sizeof(VARIABLE));
```

Figure 6 - Using 'calloc' to dynamically allocate memory.

Internal Commands

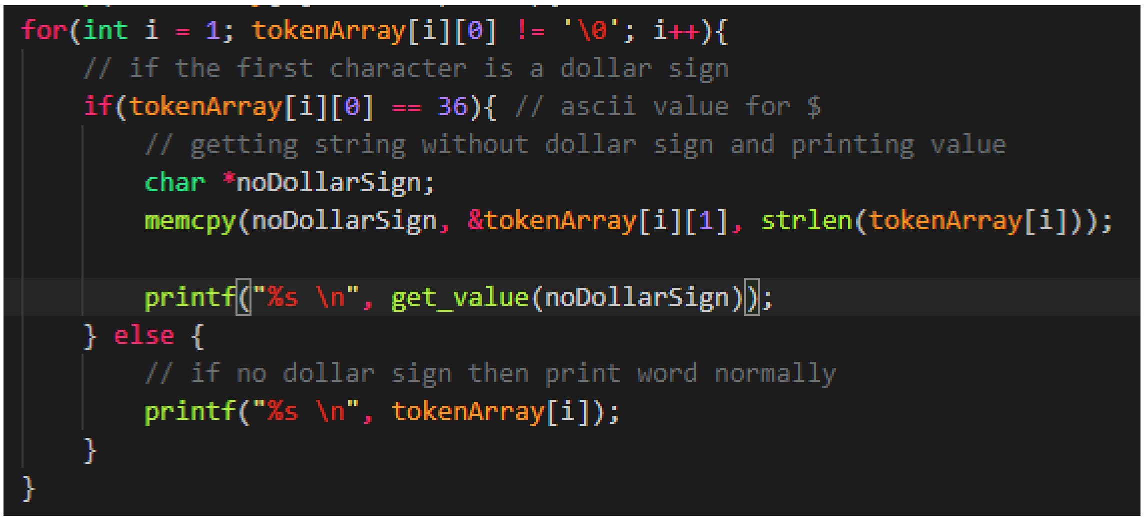
Internal commands are the hard-coded, built-in commands that the program has to offer to the user. The first one being ‘exit’, which when the user inputs “exit”, will print out an exiting message and close the program.



```c
// used to exit shell
void check_exit(char *line){
    // checking if user inputted exit
    if(line != NULL && strcmp(line, "exit") == 0){
        // exit program
        printf("\n\nExiting.\n");
        exit(EXIT_SUCCESS);
    } else {
        return;
    }
}
```

Figure 7 - Function used to exit shell.

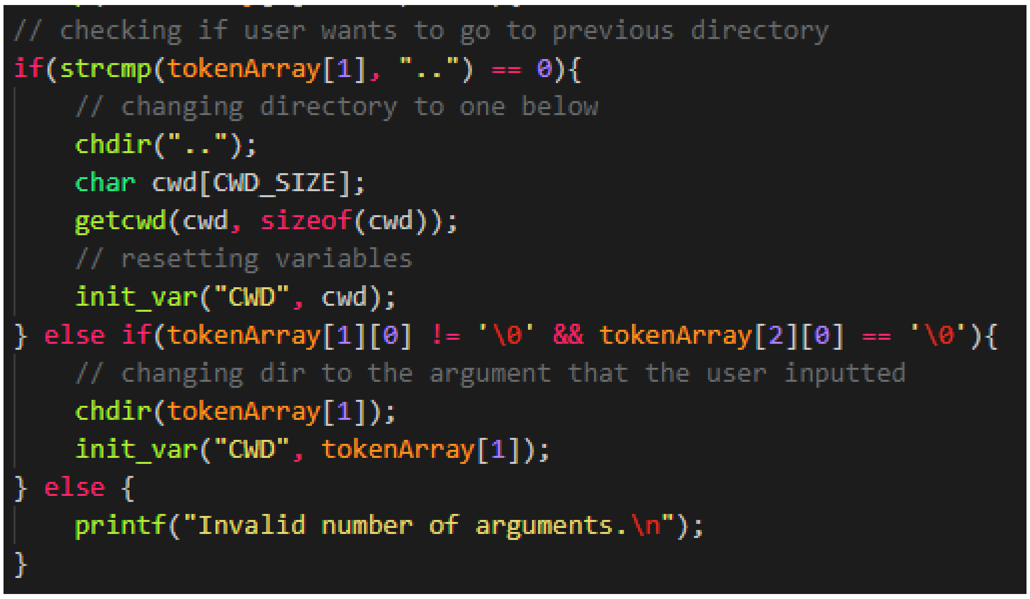
The ‘echo’ command will print what is after the keyword echo and will also print variables with the value that is in the variable as opposed to the variable name. It will check if there is a dollar sign at the beginning of each word and if there is, it will create a substring without the dollar sign and check the variable array for keys that match the one that there was in the text. If a key is matched then it will print the data of that key.



```c
for(int i = 1; tokenArray[i][0] != '\0'; i++){
    // if the first character is a dollar sign
    if(tokenArray[i][0] == 36){ // ascii value for $
        // getting string without dollar sign and printing value
        char *noDollarSign;
        memcpy(noDollarSign, &tokenArray[i][1], strlen(tokenArray[i]));

        printf("%s \n", get_value(noDollarSign));
    } else {
        // if no dollar sign then print word normally
        printf("%s \n", tokenArray[i]);
    }
}
```

Figure 8 - Function used to print text and variables.

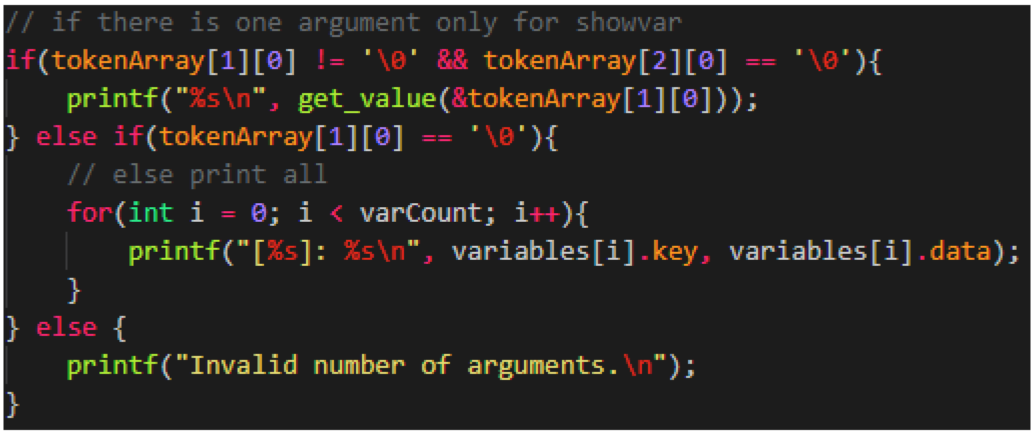
The 'cd' command is used to change directory in the shell. There are two options that the user can use. The first option is utilising the '..' command which will change the directory to the one below the current one and the second option is open to any valid directory by simply typing the directory as an argument after the cd keyword.



```
// checking if user wants to go to previous directory
if(strcmp(tokenArray[1], "..") == 0){
    // changing directory to one below
    chdir("..");
    char cwd[CWD_SIZE];
    getcwd(cwd, sizeof(cwd));
    // resetting variables
    init_var("CWD", cwd);
} else if(tokenArray[1][0] != '\0' && tokenArray[2][0] == '\0'){
    // changing dir to the argument that the user inputted
    chdir(tokenArray[1]);
    init_var("CWD", tokenArray[1]);
} else {
    printf("Invalid number of arguments.\n");
}
```

Figure 9 - Function to change directory.

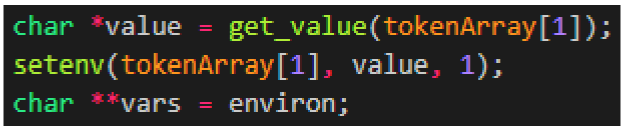
The 'showvar' function is simple, in the way that if there are no further arguments in the command, then it will just loop through the whole variable array and print them out one by one. If there is an argument after the showvar command then it will print that specific variable by searching through the array with that specific key. If found then it will print it out.



```
// if there is one argument only for showvar
if(tokenArray[1][0] != '\0' && tokenArray[2][0] == '\0'){
    printf("%s\n", get_value(&tokenArray[1][0]));
} else if(tokenArray[1][0] == '\0'){
    // else print all
    for(int i = 0; i < varCount; i++){
        printf("[%s]: %s\n", variables[i].key, variables[i].data);
    }
} else {
    printf("Invalid number of arguments.\n");
}
```

Figure 10- Function to display variables.

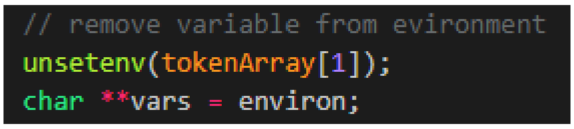
The 'export' command is used to promote a shell variable into the environmental variable scope. It is done using the built-in 'setenv' command.



```c
char *value = get_value(tokenArray[1]);
setenv(tokenArray[1], value, 1);
char **vars = environ;
```

Figure 11 - Code to promote shell variable.

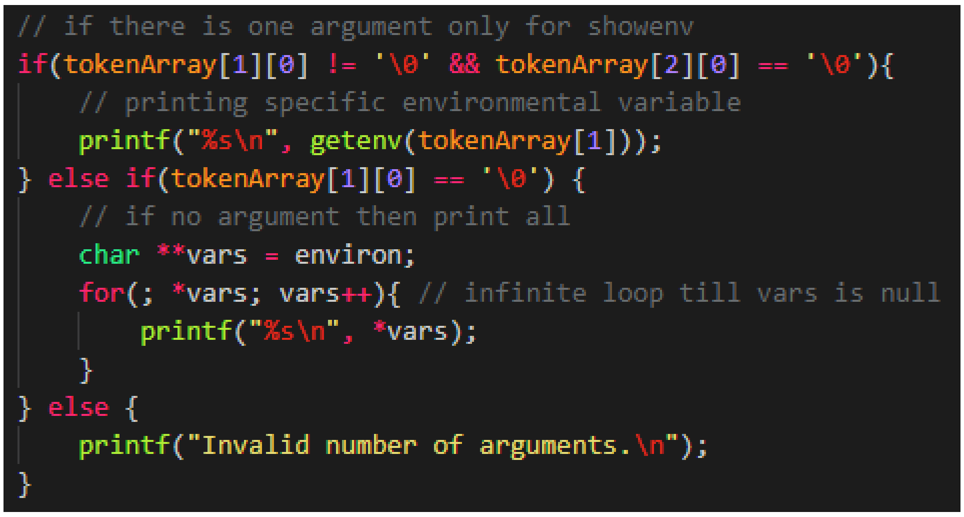
The 'unset' command is the opposite of the previous command and is used in order to delete a variable that is in the environment scope. Similarly, to the previous function it used an in-built function called 'unsetenv'.



```c
// remove variable from evironment
unsetenv(tokenArray[1]);
char **vars = environ;
```

Figure 12 - Code to delete variable.

The 'showenv' command is similar to the 'showvar' command but instead of printing all the shell variables it will print all the environmental variables and it has the same ability to print a single variable if the user specifies which variable they want after the command as an argument.

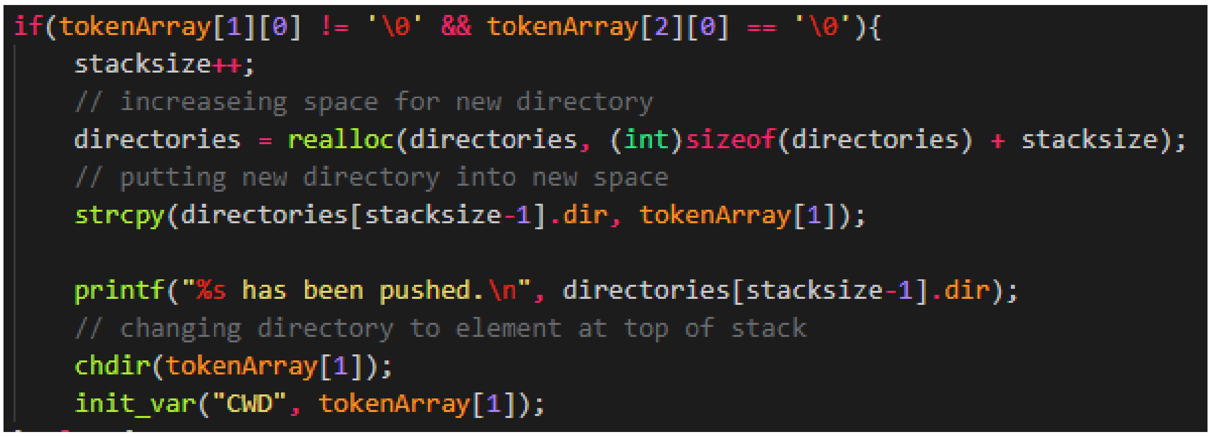


```c
// if there is one argument only for showenv
if(tokenArray[1][0] != '\0' && tokenArray[2][0] == '\0'){
    // printing specific environmental variable
    printf("%s\n", getenv(tokenArray[1]));
} else if(tokenArray[1][0] == '\0') {
    // if no argument then print all
    char **vars = environ;
    for(; *vars; vars++){ // infinite loop till vars is null
        printf("%s\n", *vars);
    }
} else {
    printf("Invalid number of arguments.\n");
}
```

Figure 13 - Function used to print all environmental varaibles.

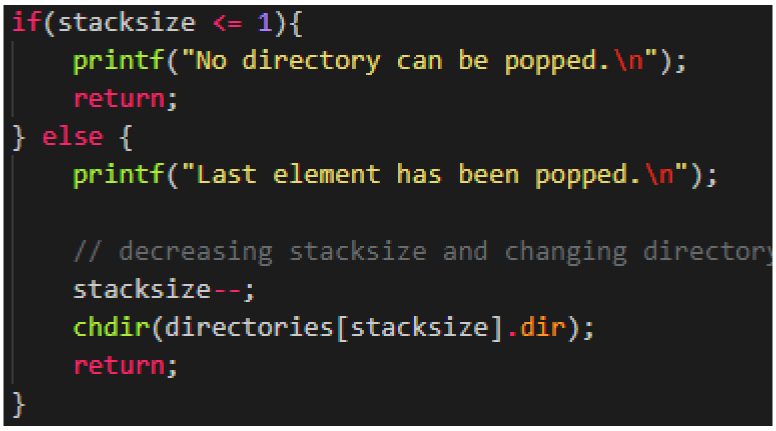
The directory stack is used to store directories whilst using a stack implementation. There are three main methods that are in play here; ‘pushd’, ‘popd’ and ‘dirs’. The first method is used for when the user wants to push a new directory to the stack. When this is done that directory will become the current working directory. The second method is used to pop the last directory that there is in the stack and return it. The last method is used to simply print all the elements in the stack one by one.

The ‘pushd’ will increase the stack size by 1 and dynamically allocate new memory for it and change the current working directory. ‘popd’ will decrease the stack size by one and change the directory also to the one below the item that was just popped. ‘dirs’ simply loops through the whole array and prints them out one by one.



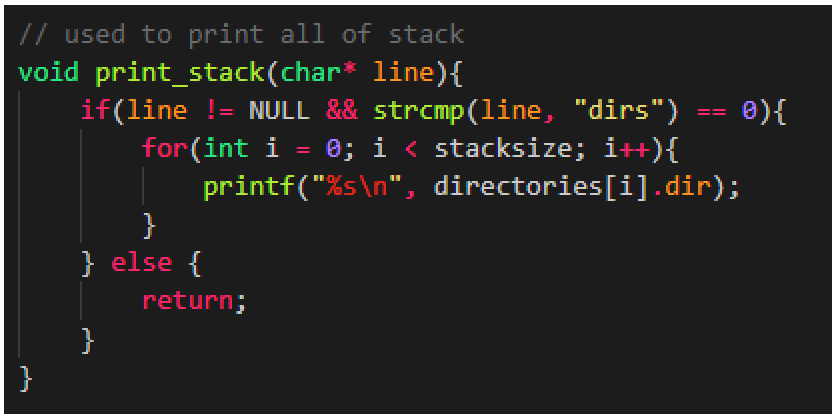
```
if(tokenArray[1][0] != '\0' && tokenArray[2][0] == '\0'){
    stacksize++;
    // increaseing space for new directory
    directories = realloc(directories, (int)sizeof(directories) + stacksize);
    // putting new directory into new space
    strcpy(directories[stacksize-1].dir, tokenArray[1]);

    printf("%s has been pushed.\n", directories[stacksize-1].dir);
    // changing directory to element at top of stack
    chdir(tokenArray[1]);
    init_var("CWD", tokenArray[1]);
```

Figure 14 - Code used to push new directory to stack.



```
if(stacksize <= 1){
    printf("No directory can be popped.\n");
    return;
} else {
    printf("Last element has been popped.\n");

    // decreasing stacksize and changing directory
    stacksize--;
    chdir(directories[stacksize].dir);
    return;
}
```

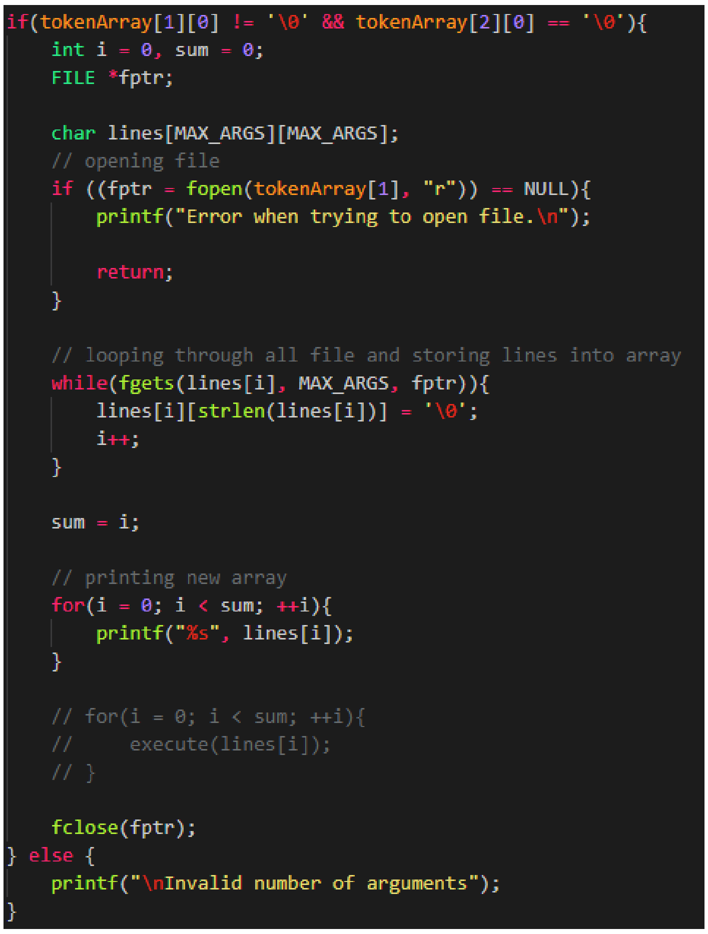
Figure 15 - Code for popping a directory.



```c
// used to print all of stack
void print_stack(char* line){
    if(line != NULL && strcmp(line, "dirs") == 0){
        for(int i = 0; i < stacksize; i++){
            printf("%s\n", directories[i].dir);
        }
    } else {
        return;
    }
}
```

Figure 16 - Function to print all directory stack.

The final internal command is the source command. This is used to take a text file as an input and read the lines one after the other and use them as inputs. Unfortunately, I did not manage to get this working completely. The lines from the file were read normally and stored into an array but the execution part was not managed.

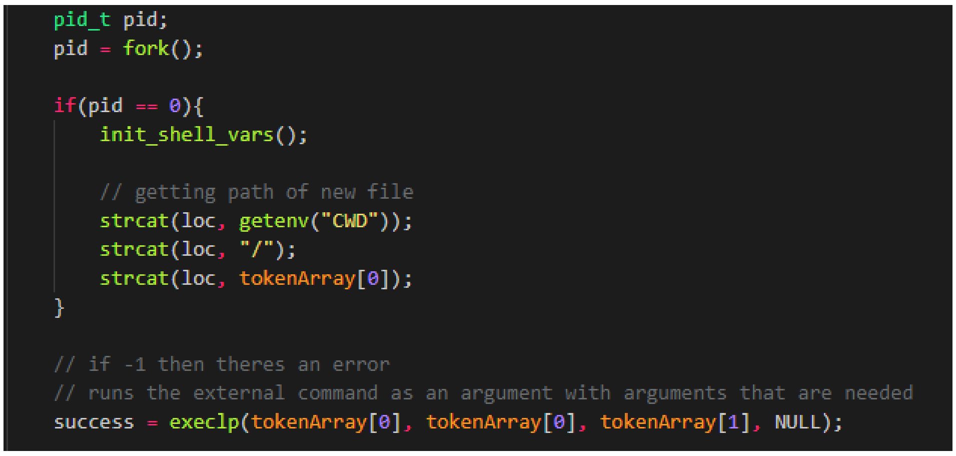


```c
if(tokenArray[1][0] != '\0' && tokenArray[2][0] == '\0'){
    int i = 0, sum = 0;
    FILE *fptr;

    char lines[MAX_ARGS][MAX_ARGS];
    // opening file
    if ((fptr = fopen(tokenArray[1], "r")) == NULL){
        printf("Error when trying to open file.\n");

        return;
    }

    // looping through all file and storing lines into array
    while(fgets(lines[i], MAX_ARGS, fptr)){
        lines[i][strlen(lines[i])] = '\0';
        i++;
    }

    sum = i;

    // printing new array
    for(i = 0; i < sum; ++i){
        printf("%s", lines[i]);
    }

    // for(i = 0; i < sum; ++i){
    //     execute(lines[i]);
    // }

    fclose(fptr);
} else {
    printf("\nInvalid number of arguments");
}
```

Figure 17 - The 'source' command code.

External Commands

This part of the algorithm is for when the command that the user inputs is not recognised by the program and thus tries to execute the command from the environment above it. This uses the fork-exec pattern which creates a process and forks it. If the process ID is zero then it will create a new path with the argument as the location. It will then run the command in the 'execlp' command.

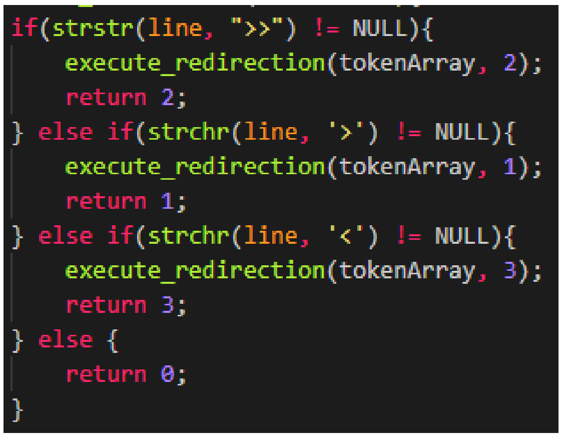


```
pid_t pid;
pid = fork();

if(pid == 0){
    init_shell_vars();

    // getting path of new file
    strcat(loc, getenv("CWD"));
    strcat(loc, "/");
    strcat(loc, tokenArray[0]);
}

// if -1 then theres an error
// runs the external command as an argument with arguments that are needed
success = execlp(tokenArray[0], tokenArray[0], tokenArray[1], NULL);
```

Figure 18 - Some code from the external commands section.

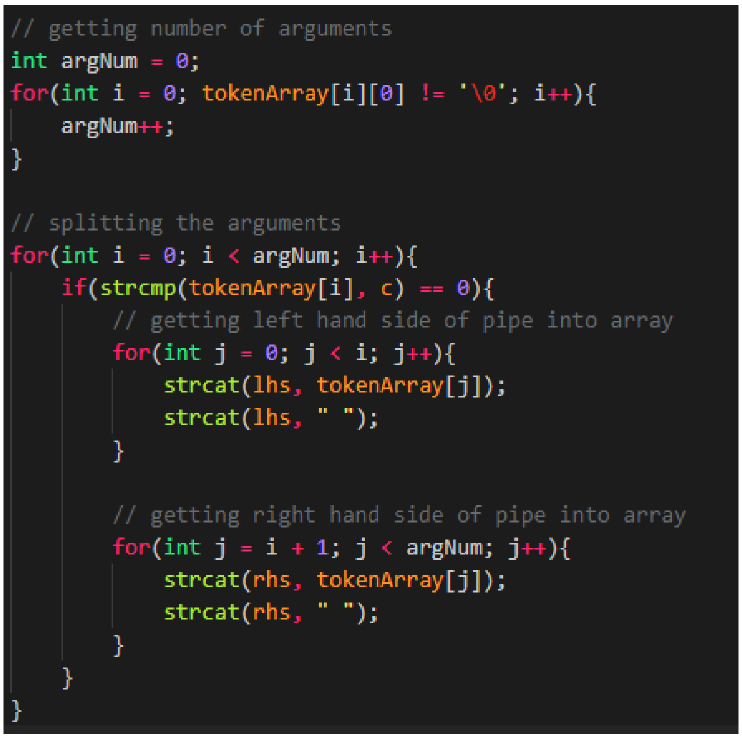
Input, Output and Redirection

This part of the program is meant to use text files as a medium for getting inputs and being able to redirect the output into a file. There was the option of appending and writing as well. Unfortunately, this part of the project is not working even though there were multiple attempts to get it working. In this case, the algorithm would first check which type of symbol is being used and then split the line with two arrays for the left and right side of the argument using the special character as the delimiter. After this the fork-exec pattern is used but there was a problem using the file descriptors that was not able to be fixed.



```c
if(strstr(line, ">>") != NULL){
    execute_redirection(tokenArray, 2);
    return 2;
} else if(strchr(line, '>') != NULL){
    execute_redirection(tokenArray, 1);
    return 1;
} else if(strchr(line, '<') != NULL){
    execute_redirection(tokenArray, 3);
    return 3;
} else {
    return 0;
}
```

Figure 20 - Method used to check was type of redirection is used.

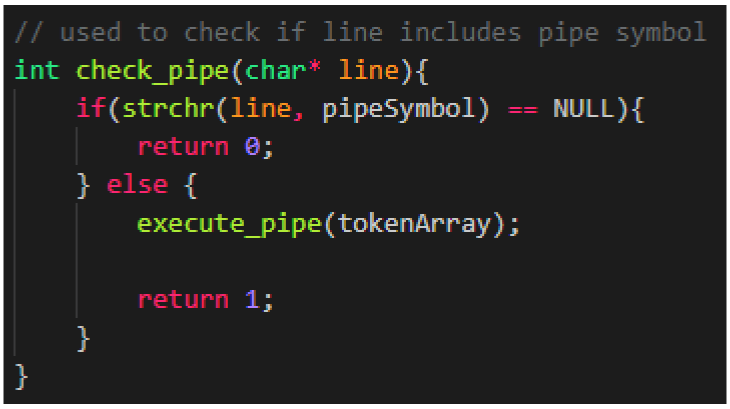


```c
// getting number of arguments
int argNum = 0;
for(int i = 0; tokenArray[i][0] != '\0'; i++){
    argNum++;
}

// splitting the arguments
for(int i = 0; i < argNum; i++){
    if(strcmp(tokenArray[i], c) == 0){
        // getting left hand side of pipe into array
        for(int j = 0; j < i; j++){
            strcat(lhs, tokenArray[j]);
            strcat(lhs, " ");
        }

        // getting right hand side of pipe into array
        for(int j = i + 1; j < argNum; j++){
            strcat(rhs, tokenArray[j]);
            strcat(rhs, " ");
        }
    }
}
```

Figure 19 - Method used in splitting arguments.

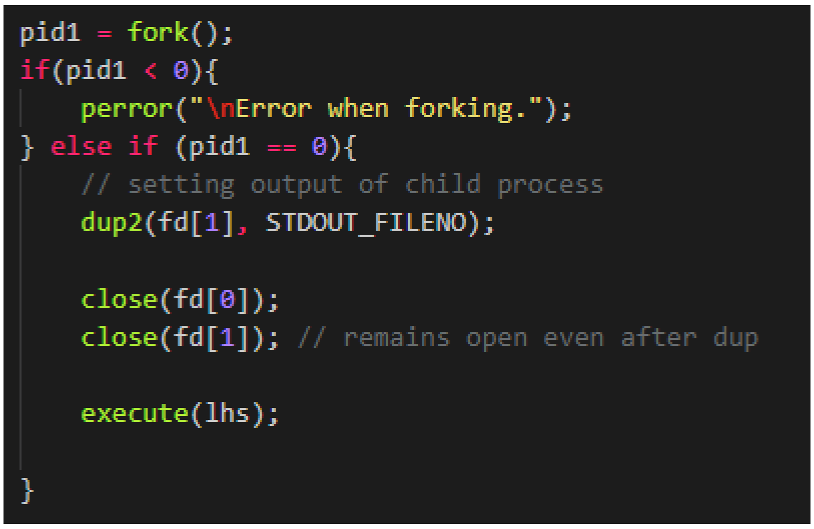
Pipelines

Piping is done when the output of one command should be the input of the next command. This way it makes chaining commands much easier and more efficient. This works similarly to the previous part of the algorithm as it uses the fork-exec method as well as splitting the argument before and after the pipe symbol. After splitting the arguments, the first command will be executed and the output will be sent to the file descriptor to match the input of the next command. In this program it is only possible to pipe two commands.



```
// used to check if line includes pipe symbol
int check_pipe(char* line){
    if(strchr(line, pipeSymbol) == NULL){
        return 0;
    } else {
        execute_pipe(tokenArray);

        return 1;
    }
}
```

Figure 21 - Checking if pipe symbol is in the text.



```
pid1 = fork();
if(pid1 < 0){
    perror("\nError when forking.");
} else if (pid1 == 0){
    // setting output of child process
    dup2(fd[1], STDOUT_FILENO);

    close(fd[0]);
    close(fd[1]); // remains open even after dup

    execute(lhs);

}
```

Figure 22 - Performing first part of the pipe.

VIDEO LINK:
https://drive.google.com/file/d/1f9l2xQfGglrnF0oUldG7mK98V0WIPtKO/view?usp=sharing