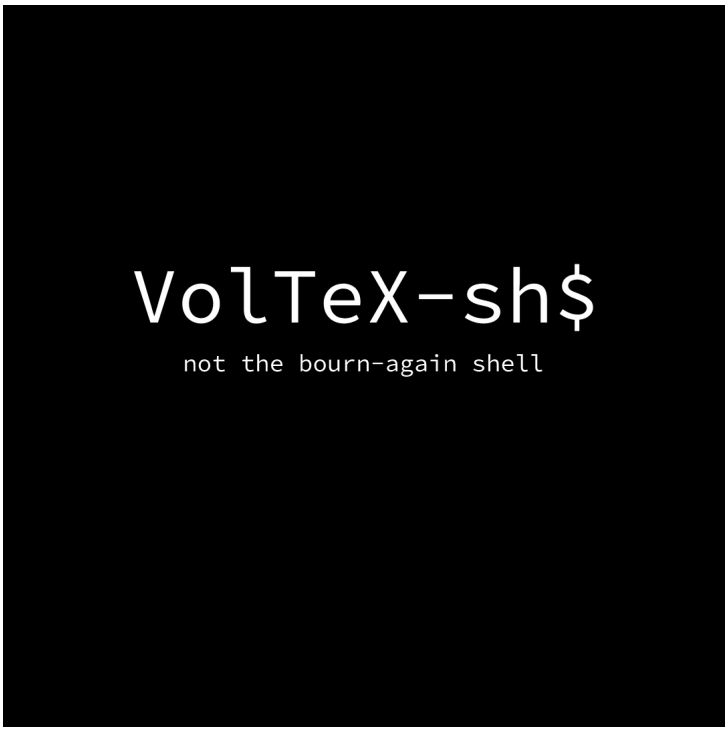
First Report
Vol-TeX-Sh

April 2021



by VolTeX Studio

BOSE Abhishek
BEKHAT Sofiane
JOHN Rajat
JY Loic Mahatsangy

Contents

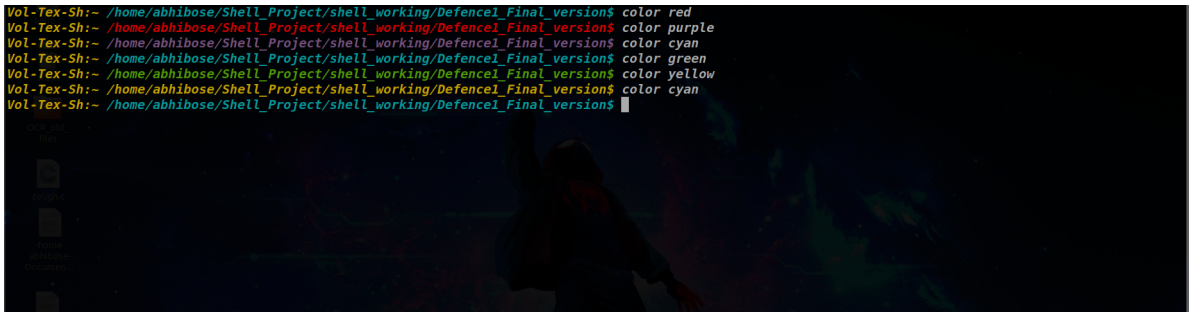
1 Overview of our Progress

1.1 Parsing

First things first, our shell needs to be able to read whatever the user types. For this end, we have to parse the input. The method is simple : we read the input and store it in an array. After, we traverse the array and extract the strings one by one and store them in another array.Furthermore, our parsing algorithm handles the case when there are multiple spaces between strings.
Feel free to read the code in section 2.1

1.2 User interface

Our shell's user interface is similar to the linux shell. The prompt displays the name of our shell followed by the path of the current directory. The color of the prompt can be changed to another color such as red, purple, yellow, cyan, blue.
Feel free to read the code in section 2.2
Here is a picture of the shell :



1.3 Execution

In order to execute a command, our shell compares the command typed by the user with all possible commands. If the comparison is successful , it executes the command by calling its function. Otherwise the shell returns an error, saying that the input command does not exist.

1.4 Commands

For our shell, we have taken heavy inspiration from the command terminal used in Linux based operating systems. All the commands that we have implemented are designed to simulate an experience similar to one you have while using a terminal.

Here is list of all the commands we have implemented:

1. pwd
2. mkdir
3. touch
4. echo
5. help
6. rmdir* (1)
7. ls* (1)
8. rm
9. cd
10. cat* (2)
11. tree
12. clear
13. color
14. mv
15. exit

(* signifies that this command also includes certain attributes that can be added to modify the behaviour of the command. Number of attributes are in the parenthesis)

1.5 Work in progress

Right now we have started the pipes and fd redirection to our shell, in addition more commands with ascii art like neofetch are to be optimised, grep too and we are working on chmod for next defence. Some networking commands are pending too.

2 Implementation

2.1 Parsing

Here is the code of the parsing algorithm :

```
void read_command(char **parameters, int *nb_par)
{
  char line[150];
int sub_index = 0;
  int c = 0;
  int counter = 0;
  int i = 0;
  while (1)
  {
    c = fgetc(stdin);
    line[counter] = (char) c;
    if (c == 10 && counter == 0)
    {
      *nb_par = 0;
      return;
    }
    if (c == 10)
      break;
    counter += 1;
  }

line[counter] = 0;

for (int j = 0; j <= counter; j++)
  {
    if (line[j] == ' ' && sub_index != 0)
    {
      parameters[i][sub_index] = 0;
i++;
sub_index = 0;
}
    else if (line[j] == 0 && sub_index != 0)
    {
      parameters[i][sub_index] = 0;
      i++;
      sub_index = 0;
    }
    else if (line[j] != ' ')
    {
parameters[i][sub_index] = line[j];
sub_index++;
    }
  }
  *nb_par = i;
```

```
  return;
}
```

2.2 User interface

Here is the code regarding the UI and the prompt :

```
void red (){
  printf("\033[0;31m");
}
void yellow(){
  printf("\033[0;33m");
}

void blue(){
  printf("\033[0;34m");
}

void purple(){
  printf("\033[0;35m");
}

void cyan(){
  printf("\033[0;36m");
}

void green () {
  printf("\033[0;32m");
}

void change_color(char *c, char *color){
      strcpy(color, c);
  }
void reset () {
  printf("\033[0m");
}

void prompt(char *color)
{
  static int f_time = 1;
  if (f_time){
    const char* screen_clear = " \e[1;1H\e[2J";
    if (write(STDOUT_FILENO, screen_clear, 12) == -1){
      fprintf(stderr, "Error while clearing the screen");
      return;
    }
    f_time = 0;
  }
  yellow();
```

```
  printf("Vol-Tex-Sh:~");
  if (strcmp(color, "yellow") == 0){
    yellow();
  }
  else if (strcmp(color, "purple") == 0){
    purple();
  }
  else if (strcmp(color, "blue") == 0){
    blue();
  }
  else if (strcmp(color, "red") == 0){
    red();
  }
  else if (strcmp(color, "green") == 0){
    green();
  }
  else{
    cyan();
  }

  char* res = pwd(1);
  printf(" %s$ ", res);
  reset();
}
```

2.3 Execution

Here is the code regarding the execution process :

```
void exec(char color[], char** parameters, int *nb_par)
{
  if (*nb_par != 0)
  {
    if (strcmp(parameters[0], "pwd") == 0){
      char* res = pwd(*nb_par);
      printf("%s\n", res);
      return;
    }
    else if (strcmp(parameters[0], "help") == 0){
      helppage(*nb_par);
      return;
    }
    else if (strcmp(parameters[0], "mkdir") == 0){
      create_dir(*nb_par, parameters);
      return;
    }
    else if (strcmp(parameters[0], "touch") == 0){
      touch(parameters, *nb_par);
      return;
```

```
    }
    else if (strcmp(parameters[0], "mv") == 0){
      mv(parameters[1], parameters[2]);
      return;
    }
    else if(strcmp(parameters[0], "rmdir") == 0){
      remove_dir(*nb_par, parameters);
      return;
    }
    else if (strcmp(parameters[0], "color") == 0){
      change_color(parameters[1], color);
      return;
    }
    else if (strcmp(parameters[0], "clear") == 0){
      clear(*nb_par);
      return;
    }
    else if (strcmp(parameters[0], "tree") == 0){
      tree(*nb_par, parameters);
      return;
    }
    else if (strcmp(parameters[0], "rm") == 0){
      _delete(*nb_par, parameters);
      return;
    }
    else if (strcmp(parameters[0], "cat") == 0){
      _cat(*nb_par, parameters);
      return;
    }
    else if (strcmp(parameters[0], "ls") == 0){
      _ls(*nb_par, parameters);
      return;
    }
    else if (strcmp(parameters[0], "cd") == 0){
      cd(*nb_par, parameters);
      return;
    }
    else if (strcmp(parameters[0], "echo") == 0){
      echo(*nb_par, parameters);
      return;
    }
    else {
      red();
      printf("SHELL : Unknown command. Type help to list all the possible com
      return;
    }
  }
}
```

2.4 The core of the shell

The core of the shell is basically an infinite loop that calls the parsing function and executes the commands. Basically, we create two processes using fork. Its a mechanism to create multiple processes. On our parent process, the shell parses the commands. But on the child process, the command is executed.
Here is the code :

```
    int main()
{
  //MAIN LOOP OF THE SHELL
  int nb_par = 0;
  char color[20];

  while (1)
  {
    char** parameters = malloc(500 * sizeof(char*));
    for (size_t i = 0; i < 500; i++)
      parameters[i] = calloc(100, sizeof(char));
    prompt(color);
    read_command(parameters, &nb_par);
    if (nb_par == 0)
    {
      free(parameters);
      continue;
    }
    if (fork() != 0)
    {
      wait(NULL);
      exit(1);
    }
    else
    {
      if (strcmp(parameters[0], "exit") == 0)
        exit(1);
      exec(color, parameters, &nb_par);
    }
    if (strcmp(parameters[0], "exit") == 0)
      exit(1);
    free(parameters);
  }
```

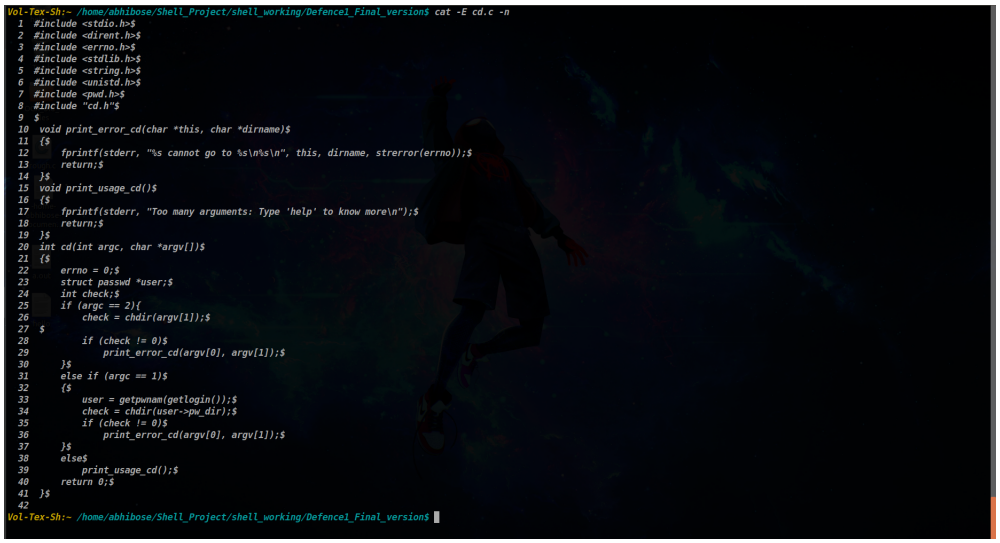
2.5 Commands

2.5.1 cat

The cat command in our shell is almost similar to what we have in bash. What it does is, prints whatever is provided on input file descriptor on Standard output. The input file descriptor could be anything, from a file to Standard input. The cat function is implemented dynamically. So the user can print multiple files at a time.

We have implemented two flags along with the default cat command.

- -E: It basically prints a dollar sign at the end of each line.
- -n: Prints the number line at the start of each line.




```
void _cat(int argc, char *argv[])
{
    int e_triggered = 0;
    int fd_in;
    int n_triggered = 0;
    int n = 1;
    int file_exists = 0;
    // Just cat command
    if (argc < 2)
    {
        re_read(STDOUT_FILENO, STDIN_FILENO, 0, &n, 0);
        return;
    }

    check_flags(argc, argv, &e_triggered, &n_triggered, &file_exists);
    if (file_exists == 0){
        re_read(STDOUT_FILENO, STDIN_FILENO, e_triggered, &n, n_triggered);
        return;
    }

    // Main loop that calls cat
    for (int i = 1; i < argc; i++)
    {
        if (strcmp("-E", argv[i]) != 0 && strcmp("-n",argv[i]) != 0 &&
        strcmp("-", argv[i]) != 0)
        {
            fd_in = open(argv[i], O_RDONLY);
            if (fd_in == -1)
            {
                fprintf(stderr, "Invalid Operand: Either file does not exists or invalid flag\n.
                Type 'help' to know more\n");
                return;
            }

            re_read(STDOUT_FILENO, fd_in, e_triggered, &n, n_triggered);
            if (i != argc - 1){
                if (write(STDOUT_FILENO, "\n", 2) == -1)
```

```
                {
                    fprintf(stderr,"Error while writing next line\n");
                    return;
                }
            }
        }
        else if (strcmp("-", argv[i]) == 0)
            re_read(STDOUT_FILENO, STDIN_FILENO, e_triggered, &n, n_triggered);
    }
    return;
}
```

Re-read is a function we wrote. Its a dynamic version of read function of stdlib.

```
void re_read(int fd_out, int fd_in, int e_triggered, int* n, int n_triggered)
{
    char buffer[Buffer_size];
    int r;
    int printed;

    if (fd_in != STDIN_FILENO && n_triggered == 1)
    {
        char number_str[50];
        printed = sprintf(number_str, "  %d  ", *n);
        if (write(fd_out, number_str, printed) == -1){
            fprintf(stderr,"Error while writing the line number\n");
            return;
        }
        *n += 1;
    }

    while (1)
    {
        r = read(fd_in, buffer, Buffer_size);
        if (r == 0)
            break;
        if (r == -1)
        {
            fprintf(stderr,"Error while reading\n");
            return;
        }
        if (fd_in == STDIN_FILENO && n_triggered == 1)
        {
            char number_str[50];
            printed = sprintf(number_str, "  %d  ", *n);
            if (write(fd_out, number_str, printed) == -1)
            {
                fprintf(stderr,"Error while writing the line number\n");
                return;
            }
            *n += 1;
        }
        rewrite(fd_in, fd_out, buffer, r, e_triggered, n, n_triggered);
    }
}
```

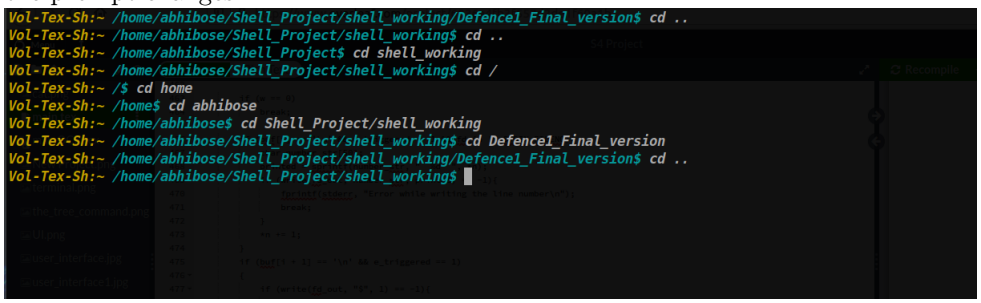
Rewrite function is again a dynamic version of write function of stdlib.

```
void rewrite(int fd_in, int fd_out, char *buf, int count, int e_triggered, int* n, int n_triggered)
{
    int i = 0;
    int printed;
    int w;
    while (1)
    {
```

```
        if (i == count)
            break;
        w = write(fd_out, &buf[i], 1);

        if (w == -1)
        {
           fprintf(stderr, "Error while writing\n");
           break;
        }

        if (w == 0)
            break;

        if (buf[i] == '\n' && n_triggered == 1 && fd_in != STDIN_FILENO){
            char number_str[50];
            printed = sprintf(number_str, "  %d  ", *n);
            if (write(fd_out, number_str, printed) == -1){
                fprintf(stderr, "Error while writing the line number\n");
                break;
            }
            *n += 1;
        }
        if (buf[i + 1] == '\n' && e_triggered == 1)
        {
            if (write(fd_out, "$", 1) == -1){
               fprintf(stderr, "Error while writing $ symbol\n");
               break;
            }
        }
        i += 1;
    }

}
```

2.5.2 cd

The cd command is again similar to what we have in bash. Changes directory. It takes in an argument. Most of the stuff for this command is self explanatory. It works something like this. You can see as we use cd to go through directories, the prompt changes.




```
    int cd(int argc, char *argv[])
{
    errno = 0;
    struct passwd *user;
    int check;
    if (argc == 2){
        check = chdir(argv[1]);
```

```
        if (check != 0)
            print_error_cd(argv[0], argv[1]);
    }
    else if (argc == 1)
    {
        user = getpwnam(getlogin());
        check = chdir(user->pw_dir);
        if (check != 0)
            print_error_cd(argv[0], argv[1]);
    }
    else
        print_usage_cd();
    return 0;
}
```

2.5.3 color

The color function changes the color of the prompt.
Here is the code :

```
void red (){
  printf("\033[0;31m");
}
void yellow(){
  printf("\033[0;33m");
}

void blue(){
  printf("\033[0;34m");
}

void purple(){
  printf("\033[0;35m");
}

void cyan(){
  printf("\033[0;36m");
}

void green () {
  printf("\033[0;32m");
}

void change_color(char *c, char *color){
      strcpy(color, c);
  }
```

2.5.4 clear

Clear command is for clearing the terminal. Kind of easy to implement.

```
void clear(int nb_par)
{
  if (nb_par > 1)
  {
    fprintf(stderr, "Too many arguments: Type '-help' to know more\n");
    return;
  }
  const char* screen_clear = " \e[1;1H\e[2J";
  if (write(STDOUT_FILENO, screen_clear, 12) == -1){
      fprintf(stderr, "Error while clearing the screen");
      return;
  }
}
```

2.5.5 echo

The echo command is used to write its arguments into output file descriptor. The output file descriptor could be Standard output or a file. It has 2 two flags to enhance the utility of this command:

- »: It appends the text on stdin to the provided file.
- >: It overwrites the text on the provided file.

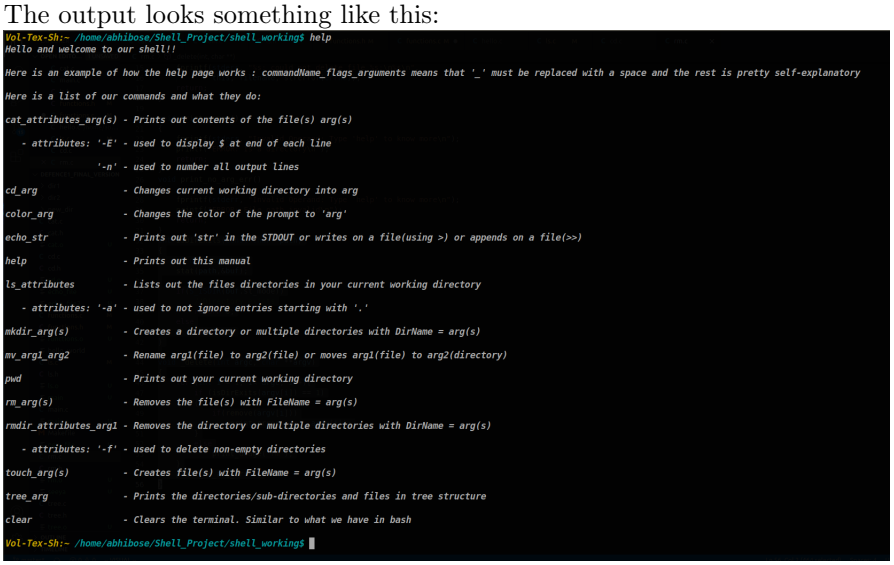
Code:

```
void echo(int nb_par, char** par)
{
  int append = 0;
  int overwrite = 0;
  int track = 0;
  for (int i = 1; i < nb_par; i++)
  {
    if (strcmp(par[i], ">>") == 0)
    {
      append = 1;
      track = i;
    }
    else if (strcmp(par[i], ">") == 0)
    {
      overwrite = 1;
      track = i;
    }
  }
  if (append == 1 && overwrite == 1)
  {
    fprintf(stderr, "Invalid arguments: Type 'help' to know more\n");
    return;
```

```
  }
  if (append == 0 && overwrite == 0)
  {
    for(int i = 1; i < nb_par; i++)
      printf("%s ", par[i]);
    printf("\n");
    return;
  }
  if(track + 1 == nb_par)
  {
    fprintf(stderr, "Token Expected: Type 'help' to know more\n");
    return;
  }
  else
  {
    if (overwrite == 1)
      _overwrite(nb_par, par, track);
    if (append == 1)
      _append(nb_par, par, track);
  }
  return;
}
```

2.5.6 help

The help command is used to print the help page with the list of all the implemented commands and instructions on how to use them into the standard output.
The output looks something like this:

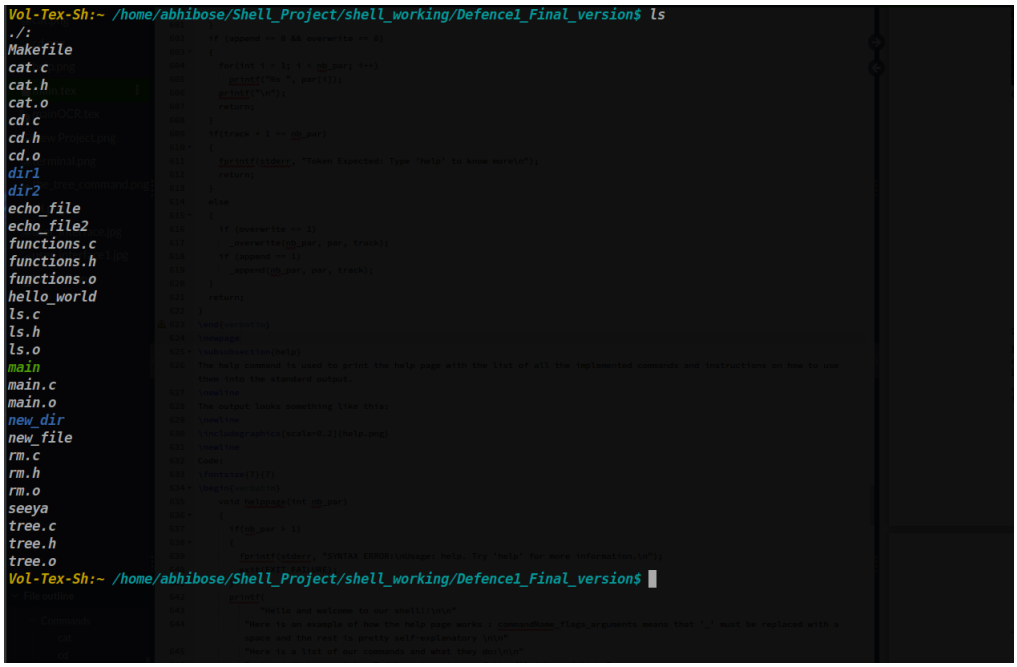


Code:

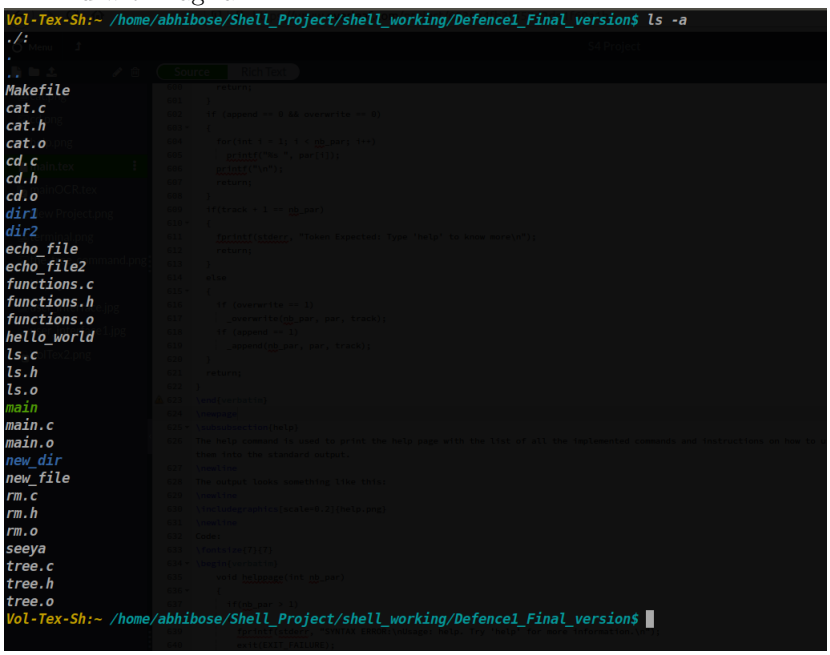
```
void helppage(int nb_par)
{
  if(nb_par > 1)
  {
    fprintf(stderr, "SYNTAX ERROR:\nUsage: help. Try 'help' for more information.\n");
    exit(EXIT_FAILURE);
  }
  printf(
       "Hello and welcome to our shell!!\n\n"
     "Here is an example of how the help page works : commandName_flags_arguments means that '_' must be replaced with a sp
     "Here is a list of our commands and what they do:\n\n"
     "cat_attributes_arg(s) - Prints out contents of the file(s) arg(s)\n\n"
     "    - attributes: '-E' - used to display $ at end of each line\n\n"
     "                  '-n' - used to number all output lines\n\n"
     "cd_arg                - Changes current working directory into arg\n\n"
     "color_arg             - Changes the color of the prompt to 'arg'\n\n"
     "echo_str              - Prints out 'str' in the STDOUT or writes on a file(using >) or appends on a file(>>)\n\n"
     "help                  - Prints out this manual\n\n"
     "ls_attributes         - Lists out the files directories in your current working directory\n\n"
     "    - attributes: '-a' - used to not ignore entries starting with '.'\n\n"
     "mkdir_arg(s)          - Creates a directory or multiple directories with DirName = arg(s)\n\n"
     "mv_arg1_arg2          - Rename arg1(file) to arg2(file) or moves arg1(file) to arg2(directory)\n\n"
     "pwd                   - Prints out your current working directory\n\n"
     "rm_arg(s)             - Removes the file(s) with FileName = arg(s)\n\n"
     "rmdir_attributes_arg1 - Removes the directory or multiple directories with DirName = arg(s)\n\n"
     "    - attributes: '-f' - used to delete non-empty directories\n\n"
     "touch_arg(s)          - Creates file(s) with FileName = arg(s)\n\n"
     "tree_arg              - Prints the directories/sub-directories and files in tree structure\n\n"
     "clear                 - Clears the terminal. Similar to what we have in bash\n\n"
}
```

2.5.7 ls

Lists files and directories. It takes in directory name otherwise the current directory. It has one flag -a or -A or –size. This flag is basically for listing out hidden files/directories.
The output looks something like this without the flag.



And with flag -a



As one can see, it is listing the hidden files in blue

```
void _list(int argc, char* argv[], int* a_triggered, int* file_dir_exists)
{
    int content_count;
    int nb_dir_in_args = 0;
    int counter = 0;
    struct dirent **contents;
    if (*file_dir_exists == 0)
```

```
    {
        if ((content_count = scandir("./", &contents, filter, alphasort)) < 0)
        {
            print_error_ls(argv[0], "./");
            return;
        }
        printf("./:\n");
        print_files_dir(*a_triggered, contents, content_count);
    }
    else
    {
        for (int i = 1; i < argc; i++)
        {
            if (strcmp(argv[i], "-a") == 0 || strcmp(argv[i], "--all") == 0
            || strcmp(argv[i], "-A") == 0)
                continue;
            nb_dir_in_args += 1;
        }


        for (int i = 1; i < argc; i++)
        {
            if (strcmp(argv[i], "-a") != 0 && strcmp(argv[i], "--all") != 0 &&
            strcmp(argv[i], "-A") != 0)
            {
                if ((content_count = scandir(argv[i], &contents, filter, alphasort)) < 0){
                    print_error_ls(argv[0], argv[i]);
                    return;
                }
                printf("%s:\n", argv[i]);
                print_files_dir(*a_triggered, contents, content_count);
                if (counter != nb_dir_in_args - 1)
                    printf("\n");
                counter += 1;
            }
        }

    }

}

int _ls(int argc, char *argv[]){
    int a_triggered = 0;
    int file_dir_exists = 0;
    int content_count;
    struct dirent **contents;

    if (argc == 1)
    {
        if ((content_count = scandir("./", &contents, filter, alphasort)) < 0)
        {
            print_error_ls(argv[0], "./");
            return 1;
        }
        printf("./:\n");
        print_files_dir(a_triggered, contents, content_count);
        return 0;
    }
    check_arguments(argc, argv, &a_triggered, &file_dir_exists);
    _list(argc, argv, &a_triggered, &file_dir_exists);
    return 0;
}
```

2.5.8 mkdir

This command is used to create one or several directories in the current working directory.
Code:

```
void create_dir(int nb_par, char** dirname)
{
  if(nb_par == 1)
  {
    fprintf(stderr, "SYNTAX ERROR:\nUsage: mkdir [dir_name].
    Try 'help' for more information.\n");
    exit(EXIT_FAILURE);
  }
  for (int i = 1; i < nb_par; i++)
  {
    int check;
    check = mkdir(dirname[i],0777);
    if (check)
        errx(-1, "There was an error creating the folder,
        maybe it already exists");
  }
}
```

2.5.9 mv

The mv command is a command that moves files from one directory to another.
Code :

```
void mv(char *file_1, char *file_2)
{
  if (rename(file_1, file_2) == -1 )
  {
    fprintf(stderr, "An error has occured during the process");
    return;
  }
}
```

2.5.10 pwd

This command is used to print out the current working directory into the standard output
Code:

```
char* pwd(int nb_par)
{
  if(nb_par > 1)
  {
    fprintf(stderr, "SYNTAX ERROR:\nUsage: pwd.
    Try 'help' for more information.\n");
    exit(EXIT_FAILURE);
  }
  char buff[FILENAME_MAX];
  char* current_working_dir = getcwd(buff, FILENAME_MAX);
  if (current_working_dir == NULL)
  {
      errx(1, "Error trying to get file");
      exit(EXIT_FAILURE);
  }
  return current_working_dir;
}
```

2.5.11 rm

It basically deletes the provided file name from the system. It is made dynamic just like we have bash.

```
    int isDirExits(const char *path)
{
    struct stat buf;
    stat(path,&buf);

    if(S_ISDIR(buf.st_mode))
        //Directory exists
        return 0;
    else
        return 1;
}

void _delete(int argc, char **argv)
{
    for (int i = 1; i< argc; i++){
        if (isDirExits(argv[i]) == 1)
        {
            if(remove(argv[i]))
                print_error("rm", argv[i]);
        }
        else
            print_usage("rm");
    }
    print_no_arg_err();
}
```

2.5.12 rmdir

This command is used to remove existing directories.
Attributes : "-f" - used to delete non-empty directories.
Code:

```
    void remove_dir(int nb_par, char** dirname)
    {
      if(nb_par == 1)
      {
        fprintf(stderr, "SYNTAX ERROR:\nUsage: rmdir [dir_name].
        Try 'help' for more information.\n");
        exit(EXIT_FAILURE);
      }
      int f = 0;
      for(int i = 1; i < nb_par; i++)
      {
        if(strcmp(dirname[i],"-f") == 0)
          {
            f = 1;
            break;
          }
      }
      for (int i = 1; i < nb_par; i++)
      {
        if(f)
        {
          if(strcmp(dirname[i],"-f") == 0)
            continue;
          DIR *theFolder = opendir(dirname[i]);
          if (theFolder == NULL)
              errx(-1, "No such directory : %s", dirname[i]);
          struct dirent *next_file;
```

```
      char filepath[FILENAME_MAX];

      while ((next_file = readdir(theFolder)) != NULL)
      {
        sprintf(filepath, "%s/%s", dirname[i], next_file->d_name);
        remove(filepath);
      }
      closedir(theFolder);
      rmdir(dirname[i]);
    }
    else
    {
      DIR *theFolder = opendir(dirname[i]);
      if (theFolder == NULL)
          errx(-1, "No such directory : %s", dirname[i]);
      closedir(theFolder);
      int check;
      check = rmdir(dirname[i]);
      if (check)
          errx(-1, "This dirrectory is not empty : %s
          (try using the -f flag)", dirname[i]);
    }
  }
}
```

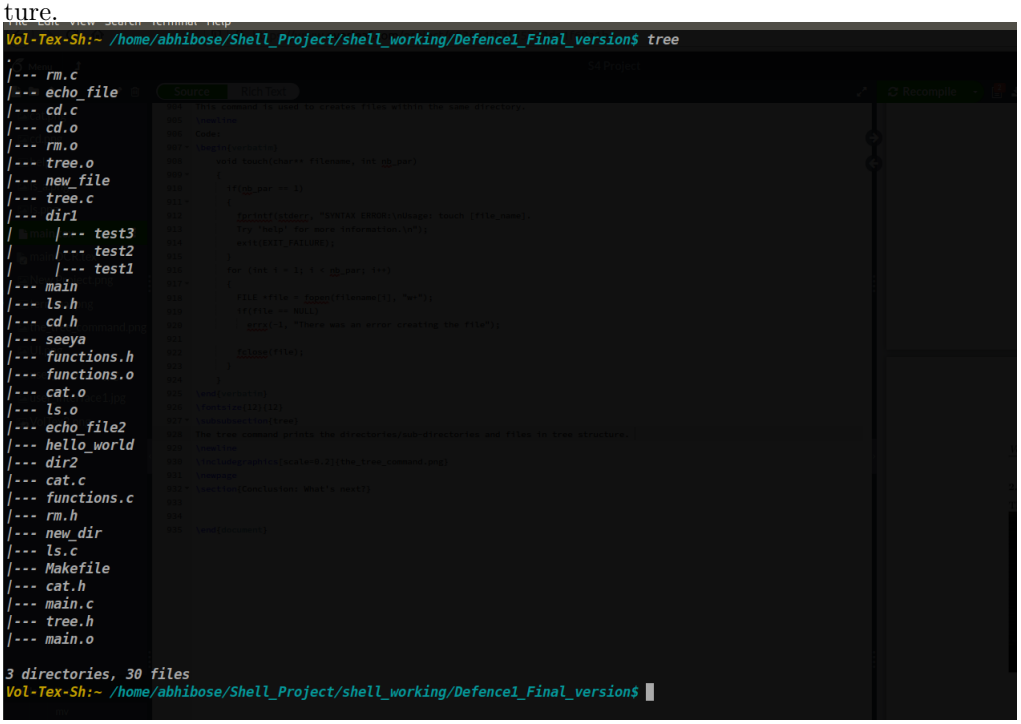
2.5.13 touch

This command is used to creates files within the same directory.
Code:

```
void touch(char** filename, int nb_par)
{
  if(nb_par == 1)
  {
    fprintf(stderr, "SYNTAX ERROR:\nUsage: touch [file_name].
    Try 'help' for more information.\n");
    exit(EXIT_FAILURE);
  }
  for (int i = 1; i < nb_par; i++)
  {
    FILE *file = fopen(filename[i], "w+");
    if(file == NULL)
      errx(-1, "There was an error creating the file");

    fclose(file);
  }
}
```

2.5.14 tree

The tree command prints the directories/sub-directories and files in tree structure.



It prints it recursively. It goes on through the directories and sub-directories. Till it doesnt find one.

```
    void PrintDirentStruct(char direntName[], int level, int* nb_dir, int* nb_file)
{
         //Define a directory stream pointer
    DIR *p_dir = NULL;

         //Define a directory structure pointer
    struct dirent *p_dirent = NULL;

         //Open the directory and return a directory stream pointer to the first directory item
    p_dir = opendir(direntName);
    if(p_dir == NULL)
    {
        my_error("opendir error");
        return;
    }

         //Read each directory item in a loop, when it returns NULL, the reading is complete
    while((p_dirent = readdir(p_dir)) != NULL)
    {
                //Directory name before backup
        char *backupDirName = NULL;

        if(p_dirent->d_name[0] == '.')
        continue;


        int i;
        for(i = 0; i < level; i++)
        {
```

```
            printf("|");
            printf("     ");
        }
        printf("|--- ");
        printf("%s\n", p_dirent->d_name);


                //If the directory item is still a directory, enter the directory
        if(p_dirent->d_type == DT_DIR)
        {
*nb_dir += 1;
                        //Current directory length
            int curDirentNameLen = strlen(direntName) + 1;

                        //Backup
            backupDirName = (char *)malloc(curDirentNameLen);
            memset(backupDirName, 0, curDirentNameLen);
            memcpy(backupDirName, direntName, curDirentNameLen);


            strcat(direntName, "/");
            strcat(direntName, p_dirent->d_name);
            PrintDirentStruct(direntName, level + 1, nb_dir, nb_file);


                        //Restore the previous directory name
            memcpy(direntName, backupDirName, curDirentNameLen);
            free(backupDirName);
            backupDirName = NULL;
        }
else
*nb_file += 1;

    }

    closedir(p_dir);
}


int tree(int argc, char* argv[])
{
char direntName[256];
memset(direntName, 0, sizeof(direntName));
int nb_dir = 0;
int nb_file = 0;
if (argc == 1)
direntName[0] = '.';
else if (argc == 2)
strcat(direntName, argv[1]);
else
    {
my_error("argument error");
        return 1;
    }

printf("%s\n", direntName);
PrintDirentStruct(direntName, 0, &nb_dir, &nb_file);
    if (nb_dir > 1)
    {
        if (nb_file > 1)
        printf("\n%i directories, %i files\n", nb_dir, nb_file);
        else
            printf("\n%i directories, %i file\n", nb_dir, nb_file);
    }
    else
    {
```

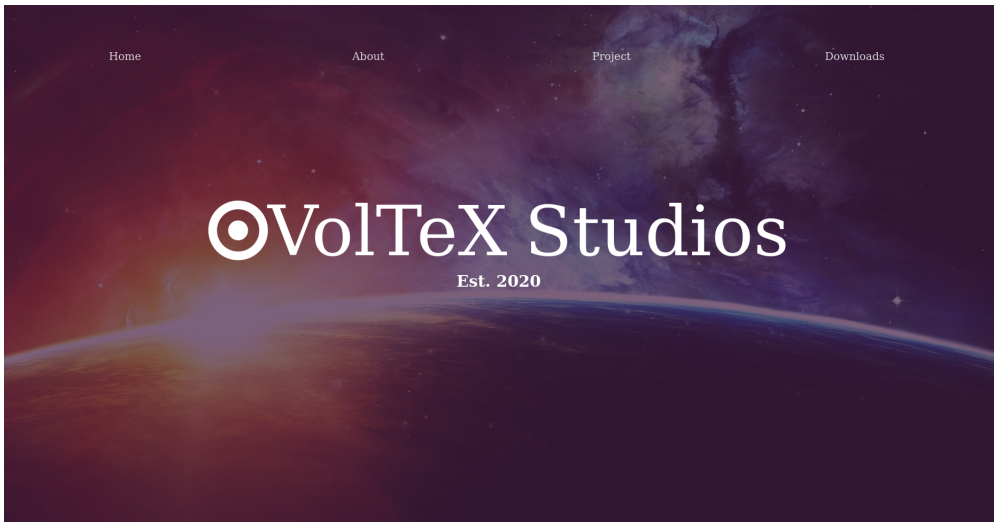
```
        if (nb_file > 1)
            printf("\n%i directory, %i files\n", nb_dir, nb_file);
        else
            printf("\n%i directory, %i file\n", nb_dir, nb_file);
    }
return 0;
}
```

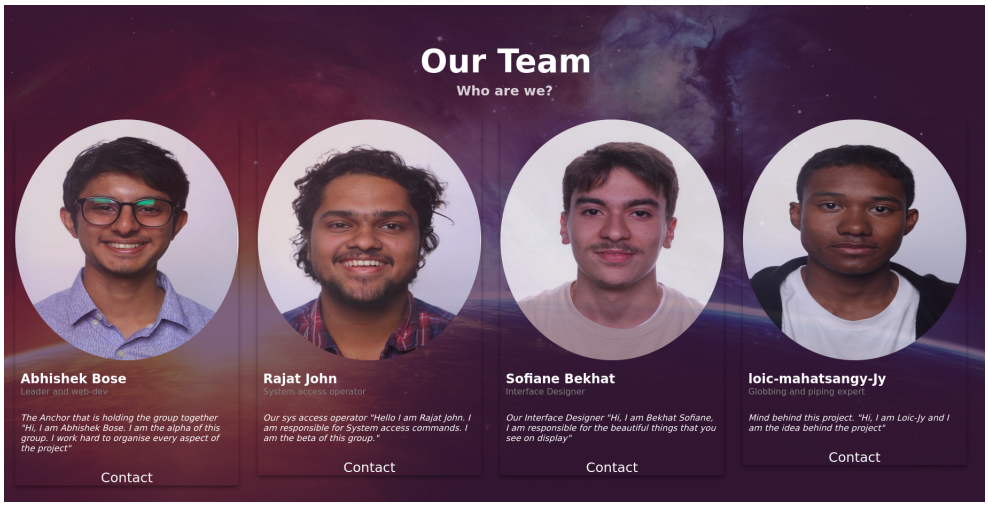
3 Website

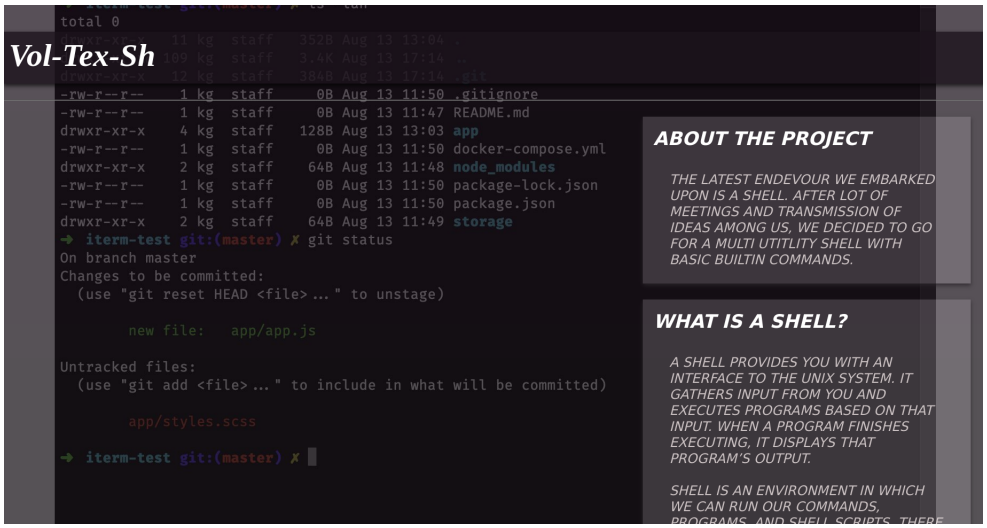
For the website we are using github pages to have it in public domain "voTeXStudios.github.io" is the URL of our website.
We have thought of 4 main sections of our website:

- Home: Which lets you back to the home page. Nothing Special.
- About: This section has two sub-sections.
  - Our Team: Basic description and contact details of our members.
  - News: Time to time updates about new stuff.
- Project: This has one sub-section which is called Vol-Tex-Sh which is basically the name of our project.
- Downloads: Links to all reports and book of specs.

This is how our wesbite looks like for now:







There are few notes to be taken that is not final version of our website. And some parts are not hyperlinked to few pages yet. We are also taking care about the fact that this website could be used in different aspect ratios and thats why we are making it responsive website.

4 What's Next??

For the next defence, we are gonna implement few more remaining builtin commands like head, tail, grep and few others. Along with that, we are already working on globbing, re-direction and auto-completion. Thinking about this is already making us excited for the next defence. We feel we have made enough progress for the first defence. One can point out that we are a bit lagging behind according to the schedule provided in the book of specs. But some of the builtin commands and the flags, were not easy to implement.
At the end, we would like to say that its a journey, we are glad we are taking.