

Shell in C

1. Introduction (What are we doing)

We are writing a shell in C. A shell is simply a program that conveniently allows you to run other programs.

2. Design

2.1. Lifetime of a shell

A shell goes by three phases in its lifetime:

1. Initialization
2. Interpret and execute
3. Terminate

Initialization phase starts with getting the current working directory and starting the **main loop**, after the initialization, the loop accepts commands, interprets them and executes them before terminating.

2.2. Main loop

So we've taken care of how the program should start up. Now, for the basic program logic: what does the shell do during its loop? Well, a simple way to handle commands is with three steps:

1. Read line
2. Tokenize line
3. Start execution

Now we've finished the design phase and understood the two main parts of the shell – its lifetime and its main loop function -, now we start implementing the shell in C.

3. Implementation

3.1. REPL (Read-Eval-Print-Loop)

We will implement a Read-Eval-Print-Loop. A REPL is a simple interactive computer programming environment that takes single user inputs, executes them, and returns the result to the user; a program written in a REPL environment is executed piecewise.

How it's called:

```
int main(int argc, char** argv)
{
    REPL();
    return 0;
}
```

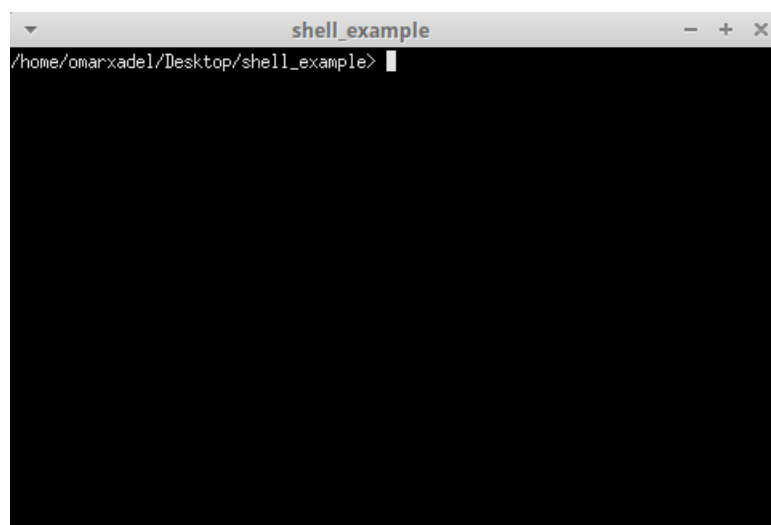
How it's defined:

```
void REPL()
{
    char* line;
    char** args;
    char* path;
    int status;

    path = (char*) malloc(PATH_SIZE*sizeof(char));

    do
    {
        if(getcwd(path, PATH_SIZE) != NULL)
        {
            printf("%s", path);
        }
        else
        {
            printf("ERROR IN GETTING WORK DIRECTORY\n");
            exit(EXIT_FAILURE);
        }
        printf("> ");
        line = read_line();
        args = tokenize_line(line);
        status = execute_cmd(args);
    }
    while(status);
}
```

The REPL gets the current working directory and prints the starting shell interface which is familiar to users as follows:



The shell then expects a user input to be entered at the `read_line()` function, which is defined as follows:

```
char* read_line()
{
    char* line = NULL;
    size_t buffersize = 0;

    if(getline(&line, &buffersize, stdin) == -1)
    {
        if(feof(stdin))
        {
            exit(EXIT_SUCCESS);
        }
        else
        {
            perror("readfile");
            exit(EXIT_FAILURE);
        }
    }
    return line;
}
```

After getting the line from the user it starts tokenizing it, i.e. converting it into pieces of command + argument 1 + argument 2 + ... till the n^{th} argument.

```
char** tokenize_line(char* line)
{
    int tok_buffsize = TOK_BFFERSIZE;
    int position = 0;
    char** tokens = (char**) malloc(tok_buffsize * sizeof(char*));
    char* token;

    if(!tokens)
    {
        printf("Allocation Error in Tokens\n");
        exit(EXIT_FAILURE);
    }

    token = strtok(line, TOK_DELIM);

    while(token != NULL)
    {
        tokens[position] = token;
        position++;

        if(position >= tok_buffsize)
        {
            tok_buffsize += TOK_BFFERSIZE;
            tokens = (char**) realloc(tokens, tok_buffsize * sizeof(char*));
            if(!tokens)
            {
                printf("Reallocation Error in Tokens\n");
                exit(EXIT_FAILURE);
            }
        }
        token = strtok(NULL, TOK_DELIM);
    }
    tokens[position] = NULL;
    return tokens;
}
```

After the tokenization is done, the shell starts the execution part which goes by 3 steps:

1- Command Validation

2- Check if exists in built-in commands then start its process

3- Process Execution

```
int execute_cmd(char** args)
{
    if(args[0] == NULL)
    {
        return 1;
    }

    for(int i = 0; i < num_builtins(); i++)
    {
        if(strcmp(args[0], builtin_str[i]) == 0)
        {
            return(*builtin_func[i])(args);
        }
    }

    return start_process(args);
}
```

The built-in commands are cd, help & exit.

```
char* builtin_str[] =
{
    "cd",
    "help",
    "exit"
};

int (*builtin_func[]) (char **) =
{
    &cmd_cd,
    &cmd_help,
    &cmd_exit
};
```

This is how each of them is defined:

```
int cmd_cd(char** args)
{
    if(args[1] == NULL)
    {
        printf("%s: Expected argument to \"cd\\\"\\n\", args[0]);
    }
    else if(args[1] != NULL && args[2] != NULL){
        printf("%s: too many arguments\\n\", args[0]);
    }
    else
    {
        if(chdir(args[1]) != 0)
        {
            printf("%s: No such file or directory\\n\", args[0]);
        }
    }
    return 1;
}

int cmd_help(char** args)
{
    int i;
    printf("Omar Adel's Shell\\n");
    printf("Enter your command and press enter to begin execution\\n");
    printf("The implemented commands are:\\n");

    for(i = 0 ; i < num_builtins() ; i++)
    {
        printf("\\t%s\\n", builtin_str[i]);
    }
    return 1;
}

int cmd_exit(char** args)
{
    return 0;
}
```

If the entered command doesn't exist in the built-ins, the shell forks a process to execute the desired command in the foreground, or might start in the background.

```
int start_process(char** args)
{
    pid_t pid, wpid;
    int status;
    int bg = 0;

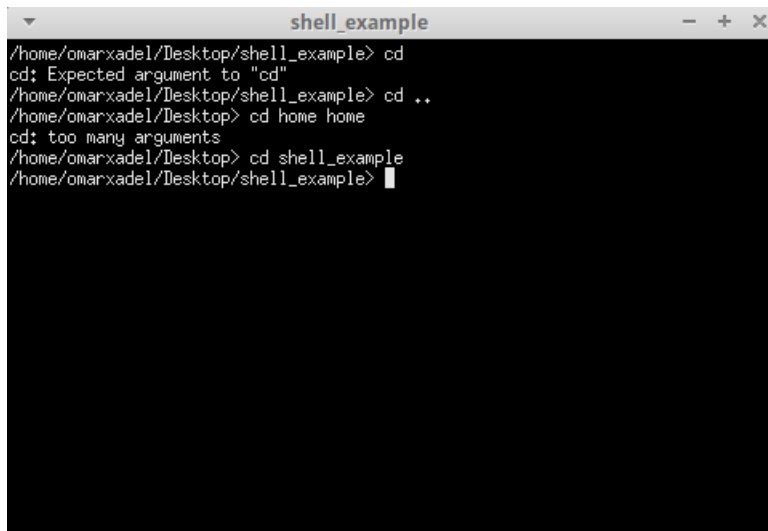
    pid = fork();

    if(args[1] != NULL && strcmp(args[1], "&") == 0) // RUN IN BACKGROUND
    {
        args[1] = NULL;
        bg = 1;
    }

    if(pid == 0)
    {
        if(execvp(args[0], args) == -1)
        {
            printf("%s: is not recognized as a command\\n\", args[0]);
        }
        exit(EXIT_FAILURE);
    }
    else if(pid < 0)
    {
        printf("ERROR FORKING\\n");
    }
    else
    {
        if(bg) // RUN IN BACKGROUND
        {
            return 1;
        }
        else // RUN IN FOREGROUND
        {
            do
            {
                wpid = waitpid(pid, &status, WUNTRACED);
            }
            while(!WIFEXITED(status) && !WIFSIGNALED(status));
            log_change(args[0], CHILD_TERMINATED);
        }
    }
}
```

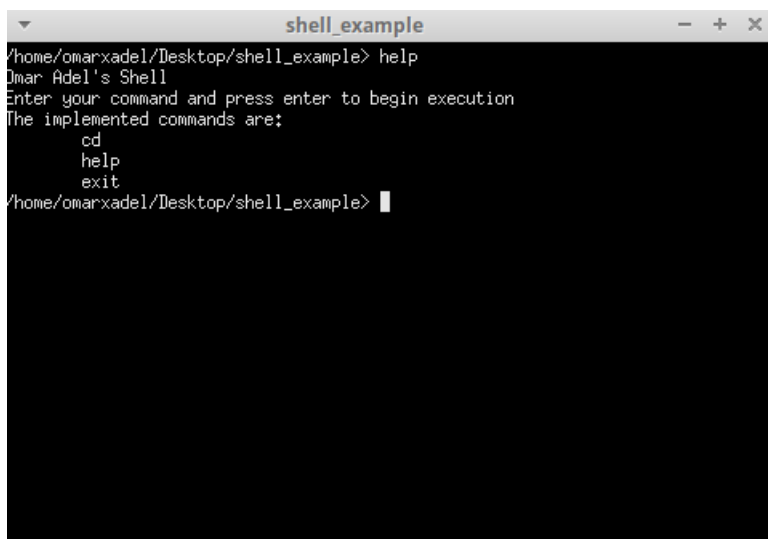
`log_change(command, change)` logs the changes in a `.log` file.

Here's a sample run for the `cd` command:



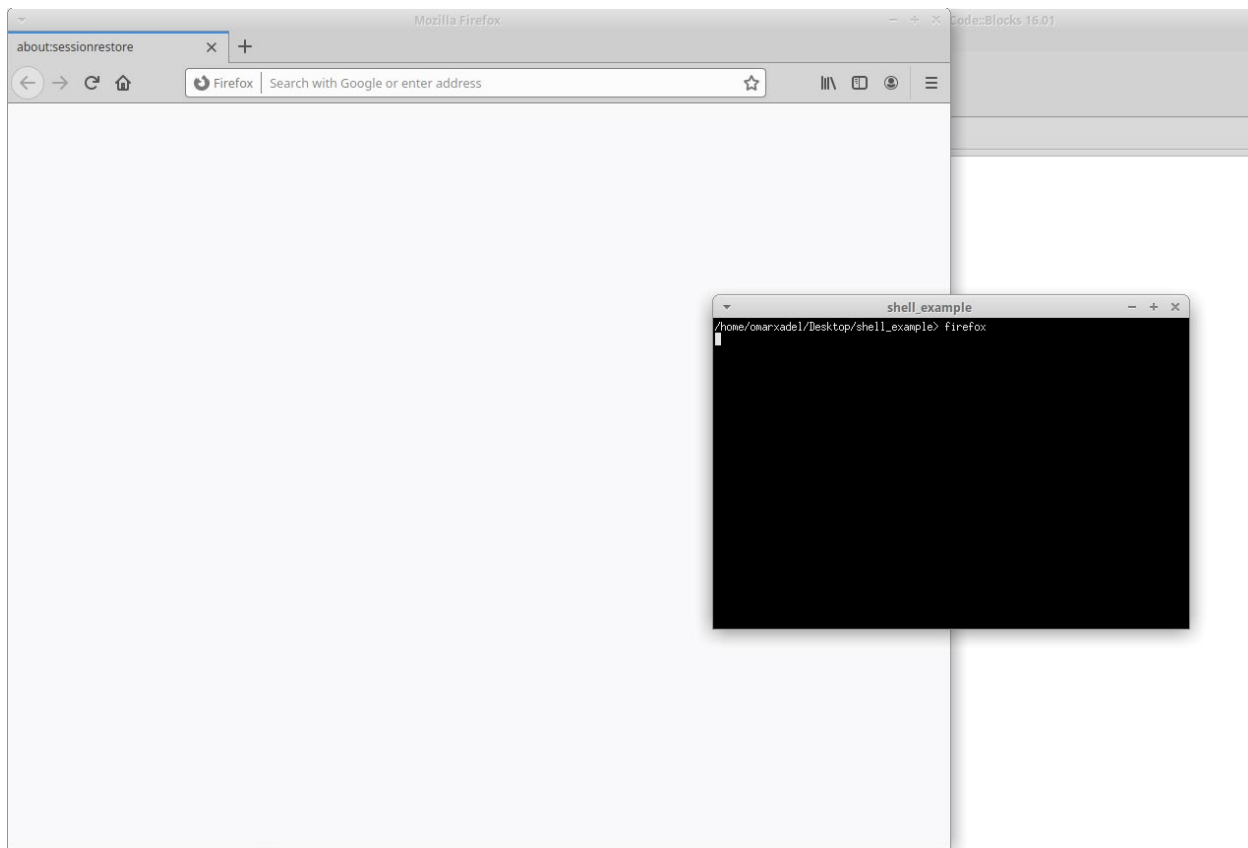
```
shell_example
/home/omarxadel/Desktop/shell_example> cd
cd: Expected argument to "cd"
/home/omarxadel/Desktop/shell_example> cd ..
/home/omarxadel/Desktop> cd home home
cd: too many arguments
/home/omarxadel/Desktop> cd shell_example
/home/omarxadel/Desktop/shell_example> █
```

Sample run for the `help` command:

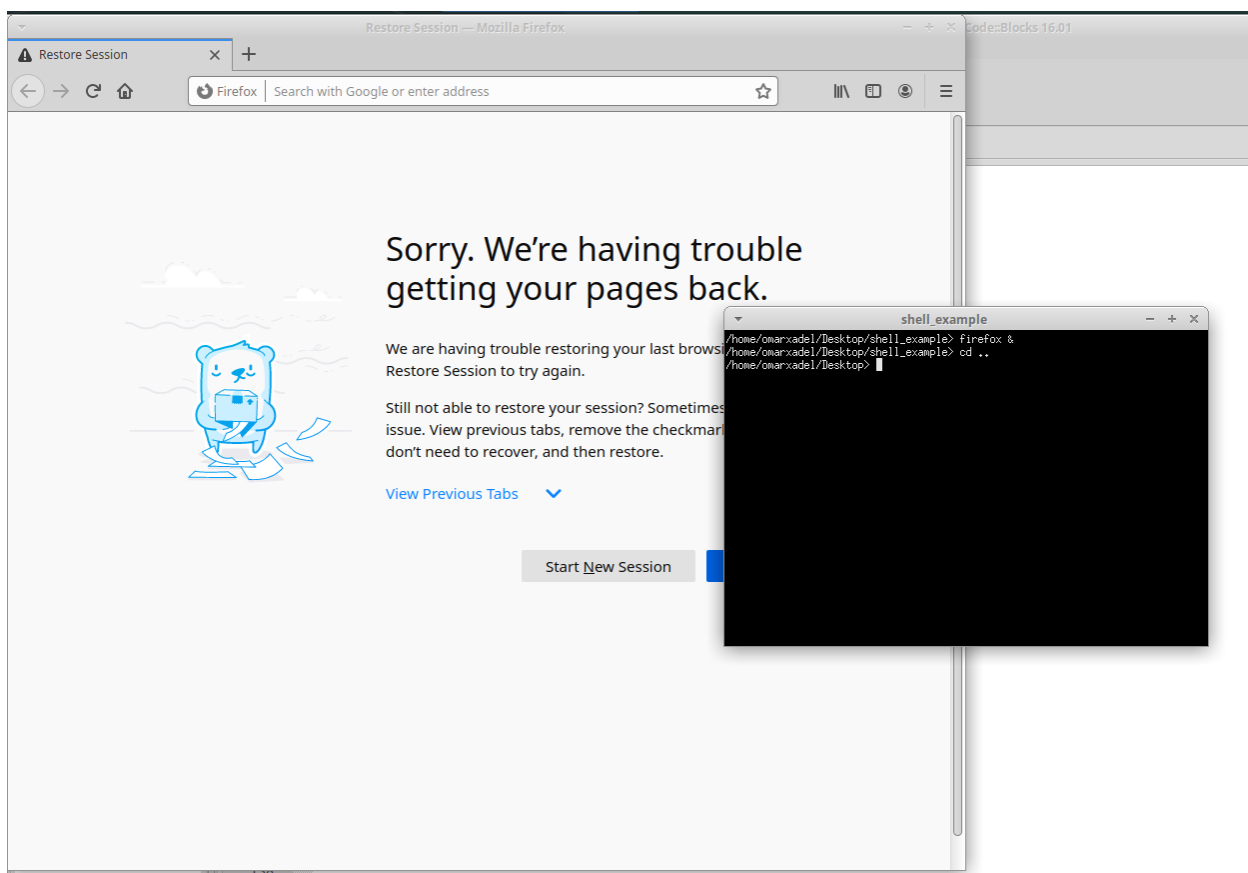


```
shell_example
/home/omarxadel/Desktop/shell_example> help
Omar Adel's Shell
Enter your command and press enter to begin execution
The implemented commands are:
  cd
  help
  exit
/home/omarxadel/Desktop/shell_example> █
```

Sample run for the `firefox` process execution in foreground:



Sample run for the firefox process execution in background:



Thank you.