Operating Systems and Systems Programming 1

Gabriel Hili

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I / We*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / our* work, except where acknowledged and referenced.

I / We* understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

GABRIEL HILI	[illegible]
Student Name	Signature
Student Name	Signature
Student Name	Signature
Student Name	Signature

CPS1012	Operating Systems and Systems Programming Report
Course Code	Title of work submitted

11/06/21
Date

Table of Contents

Abstract

In this report I explain my implementation of *smash*. There is also a link to a video presentation where I go through some code and provide some demonstrations. Afterwards, I discuss how I implemented certain features, and the limitations of the program. I also include the test cases which I used to test the program in its final version. Finally, I explain the main program, and all the functions used. To compile, navigate into the *smash* folder and type *make*. Then type *./smash* to launch.

Video

https://drive.google.com/file/d/1f0WcD_KkJCcQujM76XKc8HkSopTeTjU9/view?usp=sharing

Bugs

I could not find any bugs in the final version of the program. From my experience, I could not find any memory leaks.

Limitations

Errors are promptly handled upon violating any of the following conditions

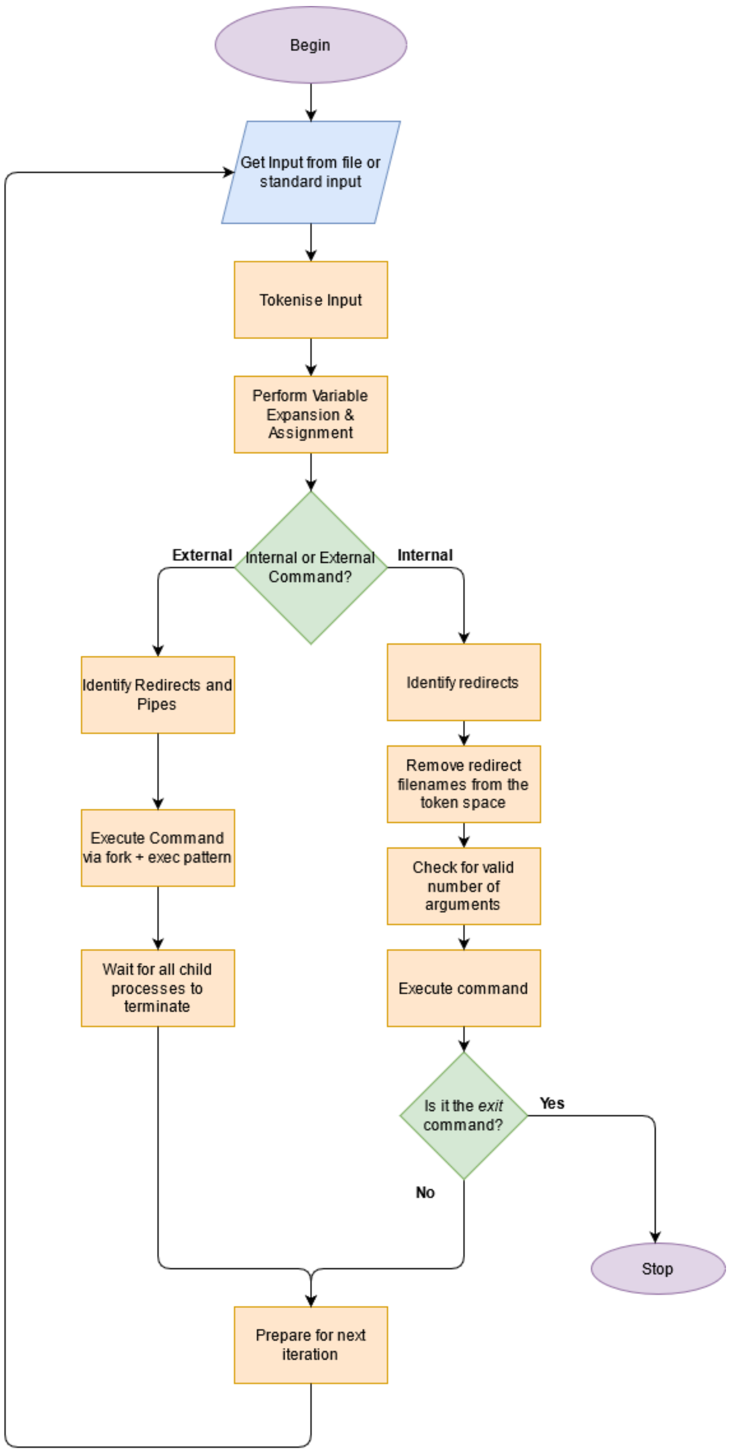
- The program only accepts commands in the following format ([…] denote a variadic nature)
  [var0=A … varN=Z cmd arg0 … argN < file0 … > fileN | …]
  Variable assignment may be omitted in the start and the user can choose to run commands with arguments or no arguments. The program still works without input/output redirects or pipes. The limitation is that the different tokens must follow this specific format's *order*. **You cannot choose to set the input/output redirects before the command, or in between the arguments.**
- *source* reads a maximum of *BUFSIZ* (as defined in *stdio.h*) characters per line.
- Directory stack may hold up to *STACK_SIZE* (as defined in *headers.h,* initially set to 100) different strings
- When reading the program executable path (line 814 in *methods.c*) it is truncated to a maximum of *BUFSIZE* (as defined in *headers.h*, initially set to 256) bytes/characters, including the *NULL* terminator.
- When reading the current working directory in (line 826 in *methods.c*) it is truncated to a maximum of *TOKEN_SIZE* (as defined in *headers.h* , initially set to 2000) character/bytes.
- As environment variables are being copied to shell variables, they are truncated to a maximum of *TOKEN_SIZE* bytes/characters.
- All tokens may contain up to *TOKEN_SIZE* characters.
- Nested source statements are not supported (It was not necessary to implement https://www.um.edu.mt/vle/mod/forum/discuss.php?d=119590)

- When executing a series of variable assignment statements, if the last assignment statement has incorrect syntax, the previous ones will still execute and be saved as shell variables.
- A maximum of *BUFSIZE* pipe characters in a pipe chain.
- A maximum of *BUFSIZE* variable assignment statements prepending a command.

Design Structure

Overall

In short, first the program gets its input either from a file or from the standard input stream. This input is then tokenised and variable expansion and assignment is performed on each token. The program then executes the command given, either as a built-in internal command, or else as an external command. The program keeps repeating until the user exits. The following flowchart is a high level representation of the general operations *smash* executes. High resolution version is attached with this report.


Begin
Get Input from file or standard input
Tokenise Input
Perform Variable Expansion & Assignment
Internal or External Command?
External
Internal
Identify Redirects and Pipes
Execute Command via fork + exec pattern
Wait for all child processes to terminate
Identify redirects
Remove redirect filenames from the token space
Check for valid number of arguments
Execute command
Is it the *exit* command?
Yes
No
Stop
Prepare for next iteration

Input

I get input from the terminal using the *linenoise* library.

Tokenising

The input string must meet the following conditions for it to be tokenised. The conditions are also re-iterated in *tokens_init* because it's the function where the majority of them are checked:

- No Redirect or Pipe Characters without a preceding token.
- No Redirect or Pipe Characters without a proceeding token.
- No Redirect or Pipe Characters not separated by a token.
- Number of Redirect Characters must not exceed BUFSIZE (as defined in *headers.h*).
- Number of Pipe Characters must not exceed BUFSIZE.
- Only the following Redirect Characters are supported. (<) (>) (>>)
- Even number of Quote Characters (").
- No Quote Characters directly adjacent to each other.
- Variable Character must be proceeded by a Normal Character or a Quote Character
- Last character cannot be an Escape Character or a Variable Character
- String must have at least one Normal Character.
- The number of Pipe Characters (ie. *ex->pipe_count*) and the number of Redirect Characters (ie. *in->redirect_count*) cannot exceed *BUFSIZE* characters.
- Only one Equal Character per token. Other instances of (=) are treated as normal
- If a token has no Equal Characters, then any other subsequent tokens will have the (=) character treated as normal (A=1 echo B=2 → outputs *B=2* and set A to 2.)

Variable expansions and assignments

During tokenisation of the input string, wherever a variable expansion character is encountered ($) the token at which the character was found, and the index of the character within that token are saved in an array. The process is similar to the variable assignment character (=).

To expand the variables, instead of looping over every character to find the variable expansion character, replacing a variable with its textual value is O(1) time since we know the exact location of the character from the array. Same goes for variable assignment.

Shell and Environment Variables

All shell variables are stored in a linked list, (the global variables *head* and *tail* are for this linked list.) Every environment variable has a shell variable counterpart. This is to make expansions and assignments easier as recommended by one of my tutors.
https://www.um.edu.mt/vle/mod/forum/discuss.php?d=118869#p195264

Any shell variable has the *env* Boolean to represent if it's also an environment variable. So a function can just check this Boolean and call *setenv* or *unsetenv* for inserting/deleting. Moreover *PATH, PROMPT, CWD, USER, HOME, SHELL, TERMINAL* and *EXITCODE* are all environment variables and are created if they do not already exist.

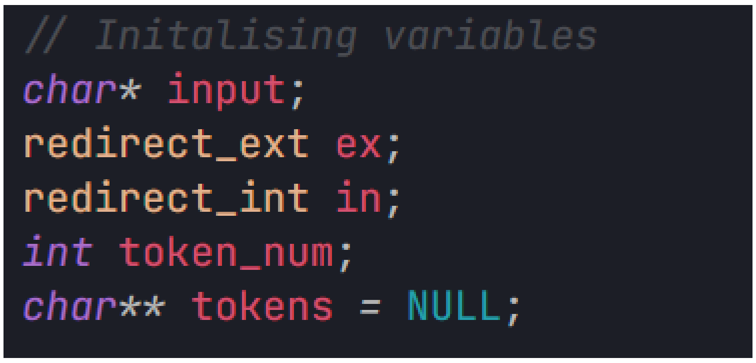
PWD does not have a shell variable counterpart. This is because *smash* uses *CWD* as both an environment and shell variable to handle the current working directory (per specifications). So any changes to the current working directory are mirrored to *PWD*(env) and *CWD*(shell and env)

tl;dr: $\mathbb{Env} \,/\, \{\, PWD \} \subseteq \mathbb{Shell}$

Redirects

I have 2 independent systems in place. One for handling redirects for internal commands, and the other for external ones. Any '*redirect*' related information (number of redirects, filenames etc …), for both internal and external commands are saved in structs of type *redirect_int* and *redirect_ext*, respectively. These structs are filled during the tokenisation process. Whenever I call the function to "execute external commands", instead of having lengthy parameters into the function, the struct is passed.

Redirection works by prepending a redirect character before the filename. As mentioned in the limitations, redirects must strictly be at the end of the input string. Redirects cannot come before the command itself or between the arguments. Moreover, in the case there are multiple redirects per line, only the last of that redirect is considered. See test cases for Redirects.

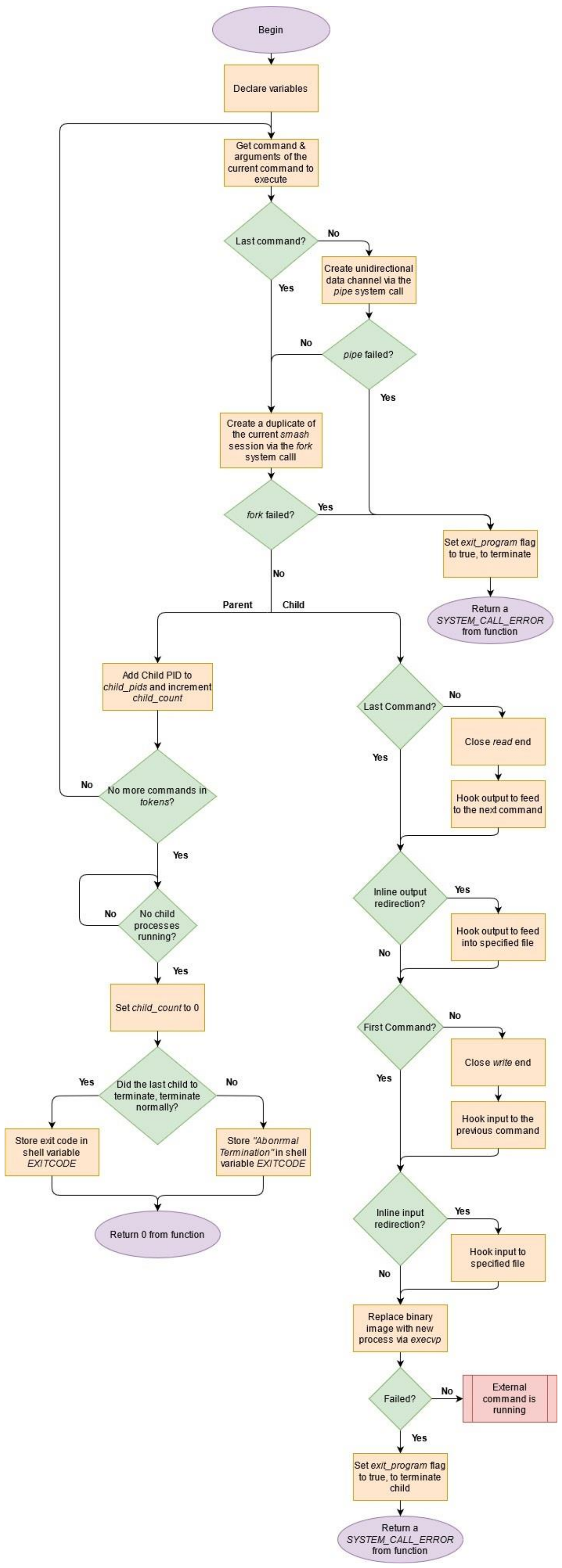


```c
// Initalising variables
char* input;
redirect_ext ex;
redirect_int in;
int token_num;
char** tokens = NULL;
```

Pipes

In a nutshell, a function loops over all the commands (I refer to them as *token chunks*) in the pipe chain, forking a child each time and execvp'ing with the command and arguments. Meanwhile the parent process keeps forking new children according to the number of *token chunks* and after it finishes iterating it waits for *all* children to terminate. If the *pipe*, *fork* or *execvp* system calls fail, the child has its *exit_program* flag set to true and will terminate gracefully, freeing all variables from the heap. The child will have its input and output hooked to the previous and next process respectively, where applicable (edge cases). The child may also decide to hook its input/output to a file instead of complying with the pipe chain. This will evidently break the pipe chain unless redirection is an input to the first command and an output to the last command. This is to mimic *bash*.

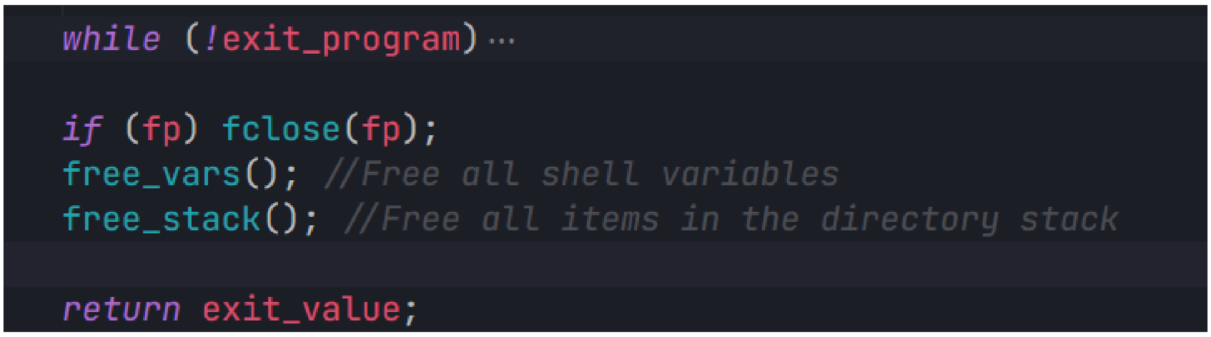
The following flowchart narrates to some more detail what I have just explained. Higher resolution version is attached with this report.


Begin
Declare variables
Get command & arguments of the current command to execute
Last command?
No
Yes
Create unidirectional data channel via the *pipe* system call
pipe failed?
No
Yes
Create a duplicate of the current *smash* session via the *fork* system calll
fork failed?
Yes
No
Set *exit_program* flag to true, to terminate
Return a *SYSTEM_CALL_ERROR* from function
Parent
Child
Add Child PID to *child_pids* and increment *child_count*
No more commands in *tokens*?
No
Yes
No child processes running?
No
Yes
Set *child_count* to 0
Did the last child to terminate, terminate normally?
Yes
No
Store exit code in shell variable *EXITCODE*
Store *"Abonrmal Termination"* in shell variable *EXITCODE*
Return 0 from function
Last Command?
No
Yes
Close *read* end
Hook output to feed to the next command
Inline output redirection?
Yes
No
Hook output to feed into specified file
First Command?
No
Yes
Close *write* end
Hook input to the previous command
Inline input redirection?
Yes
No
Hook input to specified file
Replace binary image with new process via *execvp*
Failed?
No
Yes
External command is running
Set *exit_program* flag to true, to terminate child
Return a *SYSTEM_CALL_ERROR* from function

Exiting from the program

Since the program always has some memory allocated on the heap, all of it has to be freed before the program can terminate or even just proceed to the next iteration (ie. prompting the user to enter an input after the previous input was processed). This means if the program is in a function, it cannot just call *exit()* to terminate.

Whenever the program must terminate *exit_program* is set to true and before starting a new iteration of the program, this variable is checked and if found to be true all memory in the heap is deallocated and the program terminates.

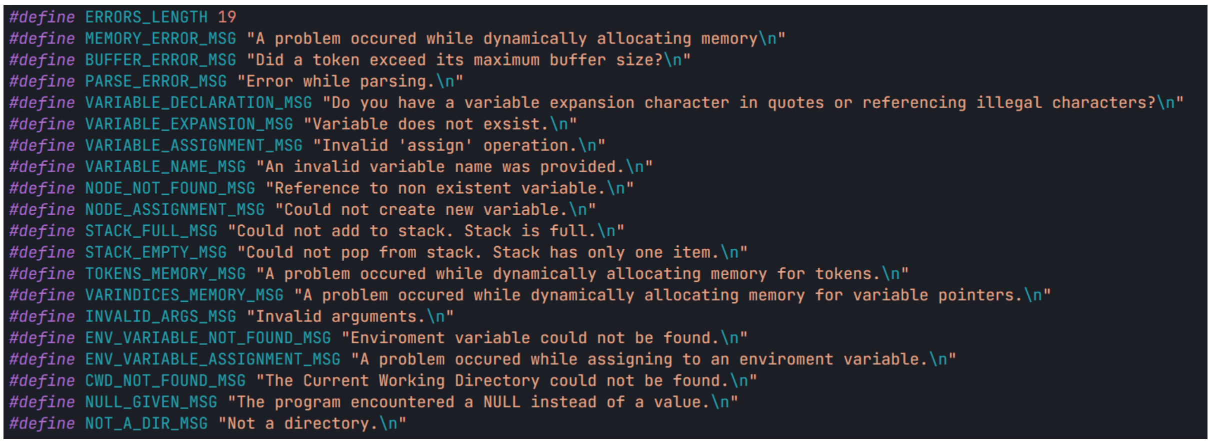


```c
while (!exit_program)···

if (fp) fclose(fp);
free_vars(); //Free all shell variables
free_stack(); //Free all items in the directory stack

return exit_value;
```

Handling errors

Let's say the use enters an unparseable string. *perror()* would not be able to output any relevant error message that *smash* does not understand the syntax of the input. Similar to how *errno* works, I created a global variable called *error* of type *err* (as defined in *headers.h*). Upon failure of any of my built-in functions, *err* is set to non-zero to indicate error. *error* also works as an index to the global array *error_msg* containing the custom error message for that particular instance of *err*.

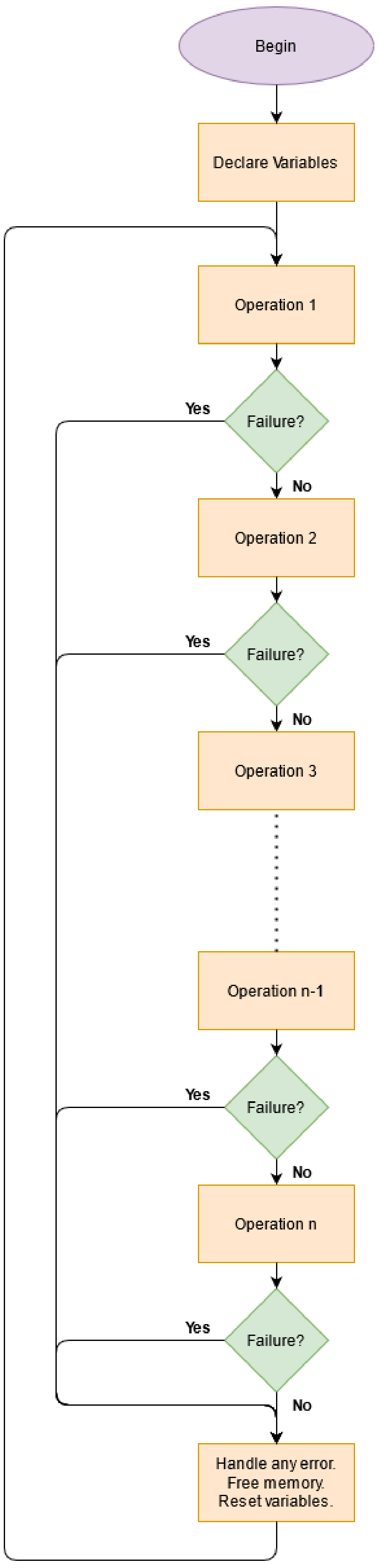
The following code snippet shows all custom error messages.



```c
#define ERRORS_LENGTH 19
#define MEMORY_ERROR_MSG "A problem occured while dynamically allocating memory\n"
#define BUFFER_ERROR_MSG "Did a token exceed its maximum buffer size?\n"
#define PARSE_ERROR_MSG "Error while parsing.\n"
#define VARIABLE_DECLARATION_MSG "Do you have a variable expansion character in quotes or referencing illegal characters?\n"
#define VARIABLE_EXPANSION_MSG "Variable does not exsist.\n"
#define VARIABLE_ASSIGNMENT_MSG "Invalid 'assign' operation.\n"
#define VARIABLE_NAME_MSG "An invalid variable name was provided.\n"
#define NODE_NOT_FOUND_MSG "Reference to non existent variable.\n"
#define NODE_ASSIGNMENT_MSG "Could not create new variable.\n"
#define STACK_FULL_MSG "Could not add to stack. Stack is full.\n"
#define STACK_EMPTY_MSG "Could not pop from stack. Stack has only one item.\n"
#define TOKENS_MEMORY_MSG "A problem occured while dynamically allocating memory for tokens.\n"
#define VARINDICES_MEMORY_MSG "A problem occured while dynamically allocating memory for variable pointers.\n"
#define INVALID_ARGS_MSG "Invalid arguments.\n"
#define ENV_VARIABLE_NOT_FOUND_MSG "Enviroment variable could not be found.\n"
#define ENV_VARIABLE_ASSIGNMENT_MSG "A problem occured while assigning to an enviroment variable.\n"
#define CWD_NOT_FOUND_MSG "The Current Working Directory could not be found.\n"
#define NULL_GIVEN_MSG "The program encountered a NULL instead of a value.\n"
#define NOT_A_DIR_MSG "Not a directory.\n"
```

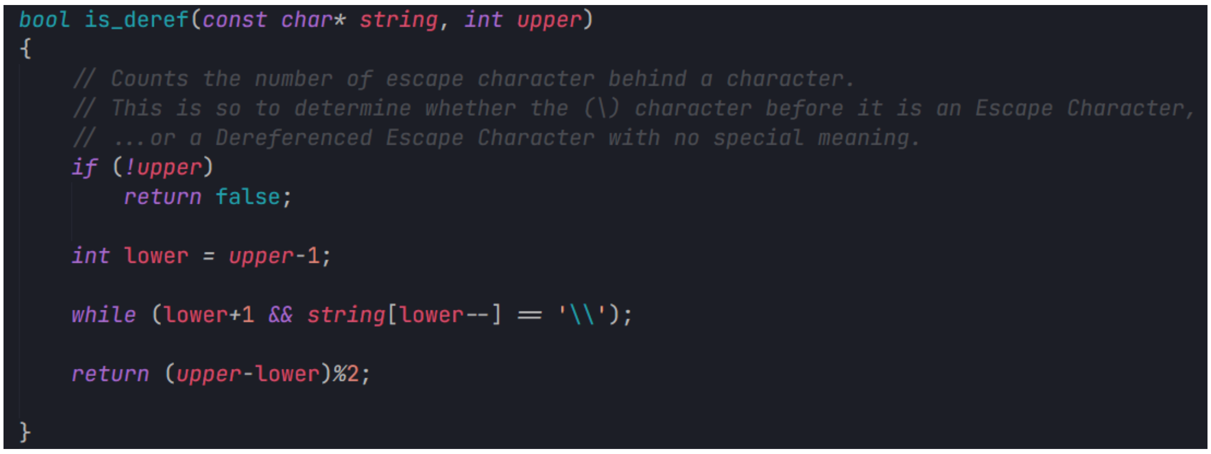
If a system call fails, *error* is set to *SYSTEM_CALL_ERROR*, which instead of printing a custom error message, *perror* is called (See *handle_error*).

If any operation in *smash.c* fails, the flow of the program is redirected to the *cleaning* phase, dictated by the label *end*. I cannot simply just call *continue* to restart the next iteration. If any function, no matter how nested it is from the calling process, returns a non-zero value, that error is propagated directly to the *end* label in *smash.c* without executing anything else in between. The following flow chart illustrates what I said.



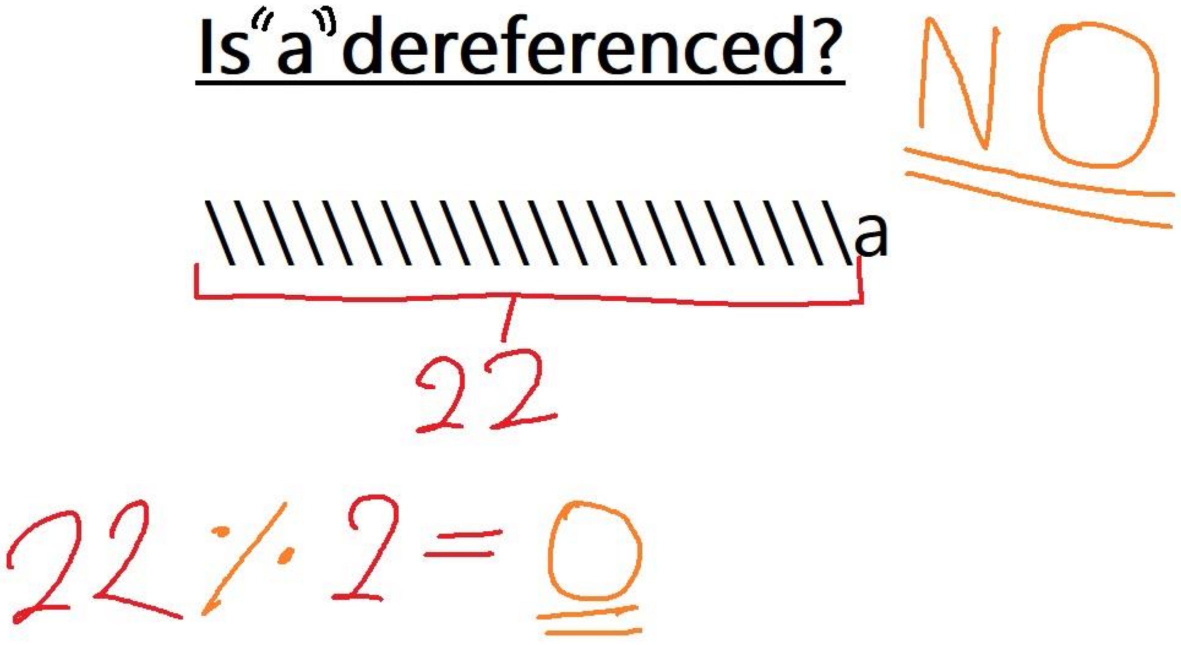
Special Mentions

I want to outline one algorithm which I think is elegant. It's how the program recognises whether a character is dereferenced or not and hence if it should hold special meaning to the shell. I quickly realised I can't just check if the previous character is a \ or not because what if that character is also dereferenced. I thought about creating an array for each character type in the string but I realised I would be creating a whole array just to only traverse through a few characters.



```c
bool is_deref(const char* string, int upper)
{
    // Counts the number of escape character behind a character.
    // This is so to determine whether the (\) character before it is an Escape Character,
    // ...or a Dereferenced Escape Character with no special meaning.
    if (!upper)
        return false;

    int lower = upper-1;

    while (lower+1 && string[lower--] == '\\');

    return (upper-lower)%2;

}
```

My solution was to not even use the character types and check the literal values. I realised that if you have an odd number of adjacent escape characters (\) then the character immediately after that string of escape characters is dereferenced. And if you have an even number, then it's not dereferenced. I think my solution is elegant, its only 4 lines of code!



Testing

I created a list of test cases and matched their results with the intended value.

Tokenising

Test Case	Expected Result	Actual result
A B C D E F G H I	[A] [B] [C] [D] [E] [F] [G] [H] [I]	Match ✓
5YMj5cDnVm jSxq0QrxLK dn12vw4OhA DryYCX98X1 H29Cx5rdqU O30gDIXeuo dgvka6QzVj jB1c9H2Rbh 5Mav4DzQWv hvrgchhq4m	[5YMj5cDnVm] [jSxq0QrxLK] [dn12vw4OhA] [DryYCX98X1] [H29Cx5rdqU] [O30gDIXeuo] [dgvka6QzVj] [jB1c9H2Rbh] [5Mav4DzQWv] [hvrgchhq4m]	Match ✓
f4P lrW fMt 87N o7d	[f4P] [lrW] [fMt] [87N] [o7d]	Match ✓
Hello\ everyone	[Hello everyone]	Match ✓
“Hello everyone”	[Hello everyone]	Match ✓
“Hello\ everyone”	[Hello everyone]	Match ✓
\”Hello Everyone\”	[“Hello Everyone“]	Match ✓
“\” Hello everyone \””	[“Hello everyone”]	Match ✓
\H\e\l\l\o\ \e\v\e\r\y\o\n\e	[Hello everyone]	Match ✓
“Hello everyone \”	Error	Match ✓
\\\\	[\\]	Match ✓
\a\/	[a/]	Match ✓
\/\/\/	[///]	Match ✓
\\\\\\	[\\\]	Match ✓
\\\\\	Error	Match ✓
“Hello $everyone”	[Hello] [$everyone]	Match ✓
“Hello everyone $”	Error	Match ✓
Hello everyone $$	Error	Match ✓
Hello everyone $\\	[Hello] [Everyone] [$\]	Match ✓
Hello everyone \$\\	[Hello] [Everyone] [$\]	Match ✓
%@+$* +**!# =$@%A &!$*@ %&&*%	[%@+$*] [+**!#] [=$@%A] [&!$*@] [%&&*%]	Match ✓
%@+$* +**!# =$@%$ &!$*@ %&&*$	Error	Match ✓
h>e<l>lo>>ever>y<on<e>!	[h] [e] [l] [lo] [ever] [y] [on] [e] [!]	Match ✓
Arrow --\>	[Arrow-->]	Match ✓
Not Arrow --\\>	[Not] [Arrow --\>]	Match ✓
>hello everyone	Error	Match ✓
Hello everyone>	Error	Match ✓
Hello>>everyone	[Hello] [everyone]	Match ✓
Hello<<everyone	Error	Match ✓
A=1 B=2 cat < a | wc –c | figlet > b	[A=1] [B=2] [cat] [a] [wc] [-c] [figlet] [b]	Match ✓

cat < a >> b	[cat] [a] [b]	Match ✓
<>	Error	Match ✓
| Hello |	Error	Match ✓
Hello |	Error	Match ✓
| Hello	Error	Match ✓
|<hello > |	Error	Match ✓
A | B | C | >	Error	Match ✓
A| B > C | D < E > F >G >> H	[A] [B] [C] [D] [E] [F] [G] [H]	Match ✓

Variable Assignment and Expansion

Test Case	**Expected Result**	**Actual Result**
NAME=One of these days	Error	Match ✓
NAME=”you can have them”	NAME = *you can have them*	Match ✓
NAME=Any\ colour\ you\ like", they are all blue."	NAME = *Any colour you like, they are all blue.*	Match ✓

NAME="Luffy" NICK=Straw\Hat NAME2="Monkey D. $NAME" DESC="$NAME2 also known as \"$NICK $NAME\" and commonly as \"$NICK\""	NAME = *Luffy*
	NICK = *Straw Hat*
	NAME2 = *Monkey D. Luffy*
	DESC = *Monkey D. Luffy also known as “Straw Hat” and commonly as “Luffy”*
		Match ✓

A=1 B=”2” C=${A}+${B}=3	A = *1*
	B = *2*
	C = *1+2=3*
		Match ✓

A=1 B=2 C=$A+$B=3	A = *1*
	B = *2*
	C = *1+2=3*
		Match ✓

A=”1” B=”2” C=”\”$A+$B=3\””	A = *1*
	B = *2*
	C = *“1+2=3”*
		Match ✓

Address=”Mr.$USER, lives at $HOME… “	Address = *Mr.gabriel_freeze, lives at /home/gabriel_freeze…*	Match ✓
Address=”Mr.$USER, lives at $HOOOMEE… “	Error (Variable does not exsist)	Match ✓
A=A=A=A=	A = *A=A=A=*	Match ✓
A==	A = *=*	Match ✓
A=A=A= A=B=B=B=	A = *B=B=B=*	Match ✓
…=A	Error	Match ✓
One1=One1	One1 = *One1*	Match ✓
1One=1One	Error	Match ✓
_1One=1One	_1One = *1One*	Match ✓
=A	Error	Match ✓
A=	Error	Match ✓
A=2 B=	A = *2* Error	Match ✓

Internal Commands

Assume the commands of each instance of a test case have no prior commands and run on a fresh instance of smash.

Test Case	**Expected Result**	**Actual Result**
A=1 B=2 echo $A+$B=5	1+3=5	Match ✓
echo “\“ \””	“ ”	Match ✓
echo	Error	Match ✓
A=1 echo A=2 A=$A	A=2 A=1	Match ✓
exit	Program returns [0] error code	Match ✓
exit 3	Program returns [3] error code	Match ✓
exit 1 2	Error	Match ✓
exit abc	Error	Match ✓
exit -3	Program returns [-3] error code	Match ✓
cd	Error	Match ✓
cd A B	Error	Match ✓
cd *<anything not a directory>*	Error	Match ✓
cd /	Current working directory is set to root.	Match ✓
cd ..	Current working directory is set to parent folder.	Match ✓
cd / cd ..	Current working directory is set to root.	Match ✓
cd $HOME	Current working directory is set to home directory, (assuming HOME has default value)	Match ✓
cd ~	Current working directory is set to home directory.	Fail ✗ *smash* does not treat ~ as referring to the home directory.
showvar	All shell variables are displayed in key-value pairs	Match ✓
A=1 B=2 C=$A showvar A B C	The values of A B C are displayed with values 1,2,1 respectively	Match ✓
A=1 B=2 D=3 showvar A B C D	A and B are displayed in key-value pairs, and then an error occurs, not displaying D.	Match ✓
A=2 export A showvar A showenv A	A is displayed twice.	Match ✓
export B	Error (Variable doesn’t exist)	Match ✓
A=1 B=2 C=3 D=4 export A D showvar A B C D showenv A D B C	All four variables are printed, then A and D are printed, then an error is raised.	Match ✓
showenv	All environment variables are displayed.	Match ✓
showenv A	Error	Match ✓

A=2 showenv A	Error	Match ✓
showenv HOME USER	HOME and USER are displayed in key value pairs.	Match ✓
unset HOME showenv HOME	Error	Match ✓
A=1 showvar A export A showenv A unset A echo $A	The value of A is displayed twice, and then a variable not found error is displayed	Match ✓
unset		Match ✓
unset HOME PATH USER showenv	HOME, PATH and USER are not included	Match ✓
A=2 B=3 unset A B C Showvar A B	2 Variable not found errors are raised	Match ✓
pushd	Error	Match ✓
pushd A B	Error	Match ✓
pushd <anything not a directory>	Error	Match ✓
pushd $HOME	CWD is changed to value of HOME. Value of HOME is added to stack	Match ✓
A=/ pushd $A	CWD is changed to root folder. Root folder is added to stack	Match ✓
pushd / popd	CWD is changed to root folder. Root Folder is removed from stack and CWD is changed to HOME directory	Match ✓
popd	Error (stack cannot be empty)	Match ✓
popd A B	Error (Invalid arguments)	Match ✓
dirs	Displays directory stack contents. Top-most items first	Match ✓
dirs A	Error	Match ✓
source	Error	Match ✓
source *<file that doesn't exist>*	Error	Match ✓
source test.txt	Reads commands line by line from *test.txt*	Match ✓
source *<file with the source command included>*	Executes up until the source command, then outputs an error and terminates source command	Match ✓
source *<file with command that appends to itself>*	Program reads appended lines.	Match ✓
source *<file with command that writes to itself>*	Program reads all lines of the old file, then is overwritten	Match ✓

source <file with command that takes input from itself>	The command takes the whole file as an input, including the command itself.	Match ✓
source <file with command that takes input from itself and outputs a command to itself>	File grows exponentially, and source does not break.	Match ✓ Can only be stopped by sending SIGKILL to smash

Redirects

Test Case	**Expected Result**	**Actual Result**
echo hello > a	Nothing is displayed. *a* is created/truncated with 'hello'	Match ✓
echo hello > a < b	Nothing is displayed. *a* is created/truncated with 'hello'	Match ✓
echo hello >> a	Nothing is displayed. *'hello'* is appended to the current contents of *a.*	Match ✓
echo hello > a >> a > a >> a	Nothing is displayed. '*hello'* is appended to the current contents of *a.*	Match ✓
echo hello > a >> a > a >> a > a	Nothing is displayed. *a* is created/truncated with 'hello'	Match ✓
echo hello > a >> b > c >> d > e	Nothing is displayed. *e* is created/truncated with 'hello'	Match ✓
echo hello >a >b >c >d >e	Nothing is displayed. *e* is created/truncated with 'hello'. Other files are not considered.	Match ✓
echo hello < a > b >> c	Nothing is displayed. '*hello'* is appended to the current contents of *c.*	Match ✓
A=1 | B=2	Error	Match ✓
A=1 < B=2 > C=3	Error	Match ✓
A = 1 | B = 2	Program should look for command *A* and pass arguments = and *1*. Similarly for B	Match ✓

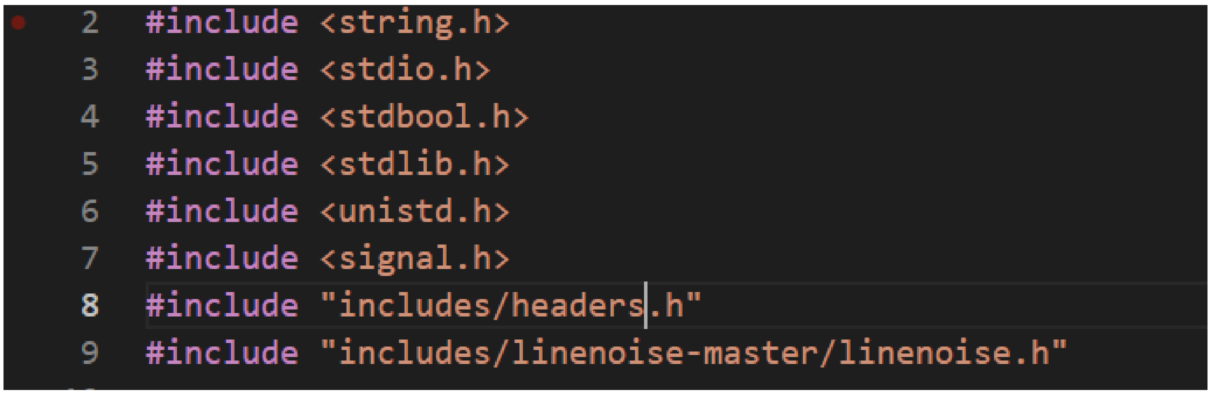
External Commands

Test Case	**Expected Result**	**Actual Result**
<doesn't exist> | cat file | wc -c | figlet	Error. *cat* receives no input but the pipe chain executes regardless. 0 in *figlet* should be outputted	Match ✓
cat file | wc –c | figlet | <doesn't exist>	Error. Nothing is displayed	Match ✓
cat file | wc –c | figlet > file2 | <doesn't exist>	Error. Nothing is displayed. *file2* has the *figlet* number of characters of *file*.	Match ✓

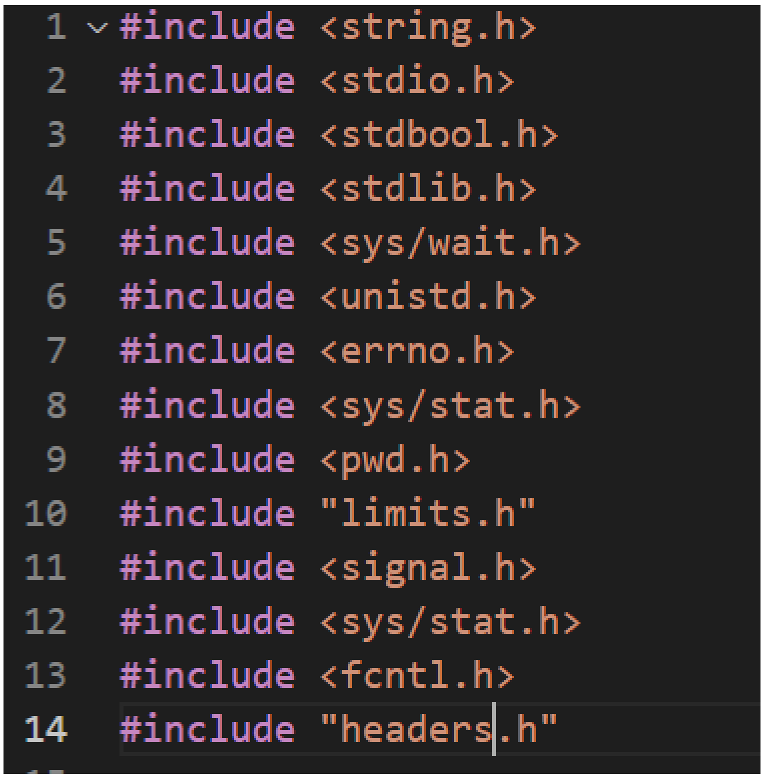
cat < file > file 2 | wc –c | figlet	*wc-c* receives no input, but pipe chain executes regardless. 0 in *figlet* is outputted and *file2* has the contents of *file*	Match ✓
cd smash/smash-2 ./smash	*smash* is relaunched	Match ✓
echo Hello | figlet |ls	*ls* executes as if alone	Match ✓

Libraries used

smash.c makes use of the following external dependencies


```c
#include <string.h>
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include "includes/headers.h"
#include "includes/linenoise-master/linenoise.h"
```

methods.c makes use of the following external dependencies

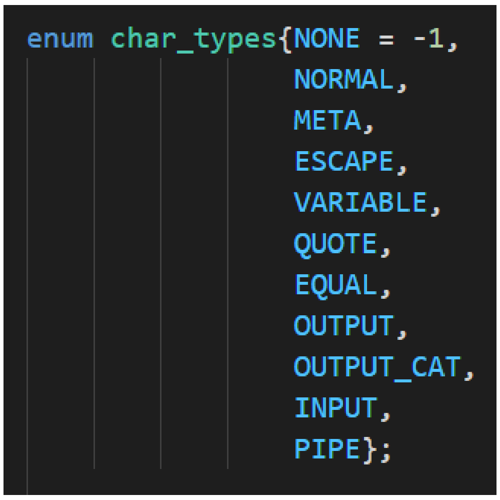

```c
#include <string.h>
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <sys/stat.h>
#include <pwd.h>
#include "limits.h"
#include <signal.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "headers.h"
```

Headers

In this .h file there are the function protocols of *methods.c* and definitions of pre-processor directives and global variables.

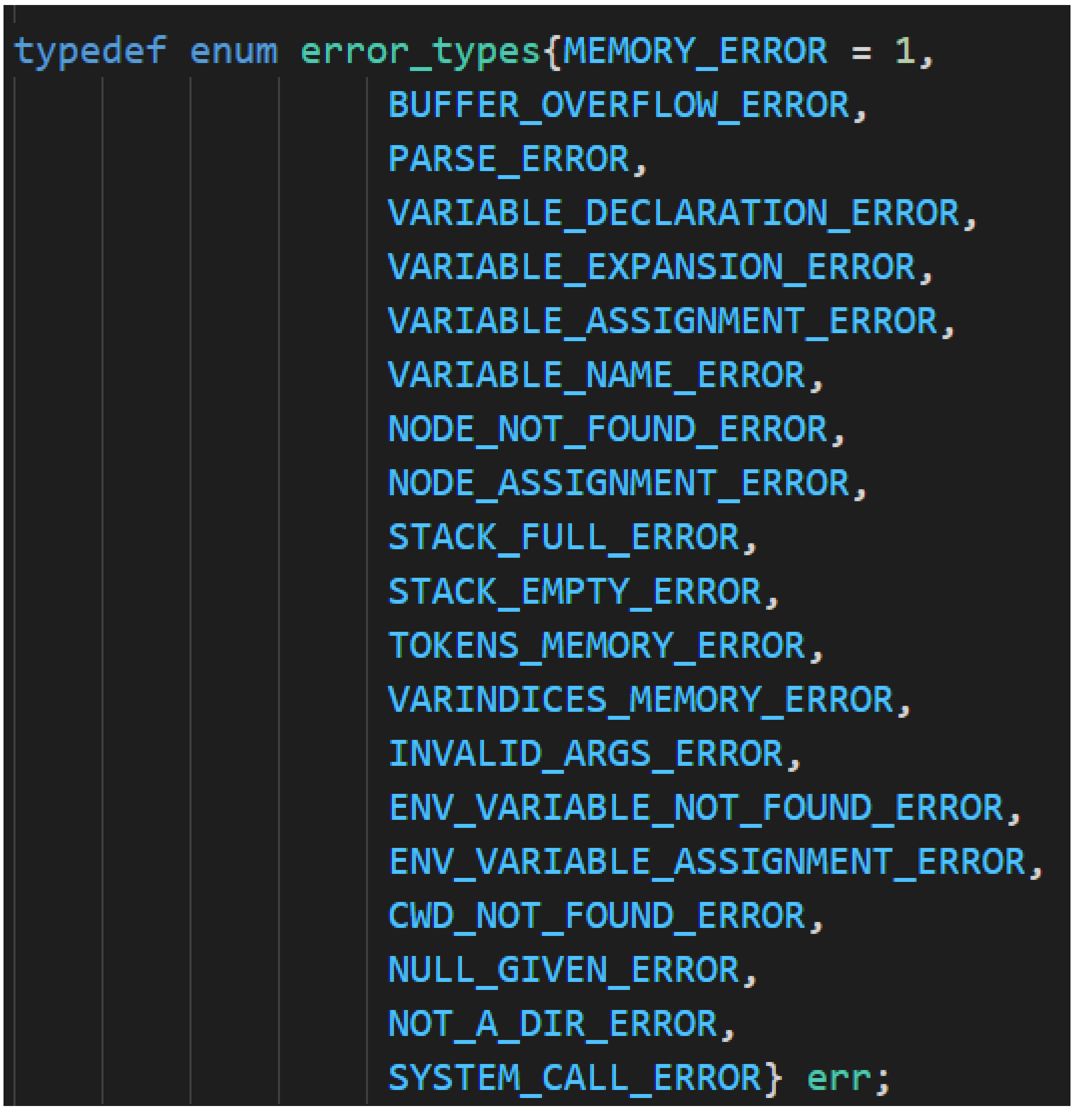
BUFSIZE is used to declare arrays with a small length

char_types is used to distinguish between different characters. This is used for tokenising the input.



```
enum char_types{NONE = -1,
                NORMAL,
                META,
                ESCAPE,
                VARIABLE,
                QUOTE,
                EQUAL,
                OUTPUT,
                OUTPUT_CAT,
                INPUT,
                PIPE};
```

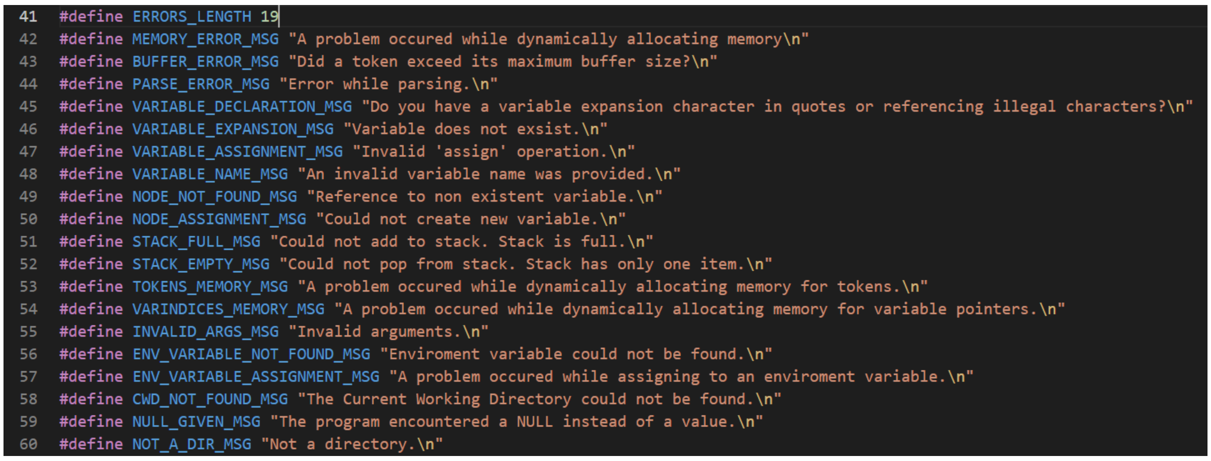
An instance of *err* is returned by functions in case of an error. *handle_error()* uses these definitions as an index to the array *errors* to display the appropriate error message. Notice how there is no definition for 0, because returning 0 from functions means there was no error.



```c
typedef enum error_types{MEMORY_ERROR = 1,
                        BUFFER_OVERFLOW_ERROR,
                        PARSE_ERROR,
                        VARIABLE_DECLARATION_ERROR,
                        VARIABLE_EXPANSION_ERROR,
                        VARIABLE_ASSIGNMENT_ERROR,
                        VARIABLE_NAME_ERROR,
                        NODE_NOT_FOUND_ERROR,
                        NODE_ASSIGNMENT_ERROR,
                        STACK_FULL_ERROR,
                        STACK_EMPTY_ERROR,
                        TOKENS_MEMORY_ERROR,
                        VARINDICES_MEMORY_ERROR,
                        INVALID_ARGS_ERROR,
                        ENV_VARIABLE_NOT_FOUND_ERROR,
                        ENV_VARIABLE_ASSIGNMENT_ERROR,
                        CWD_NOT_FOUND_ERROR,
                        NULL_GIVEN_ERROR,
                        NOT_A_DIR_ERROR,
                        SYSTEM_CALL_ERROR} err;
```

ERRORS_LENGTH refers to the length of the array *errors*.

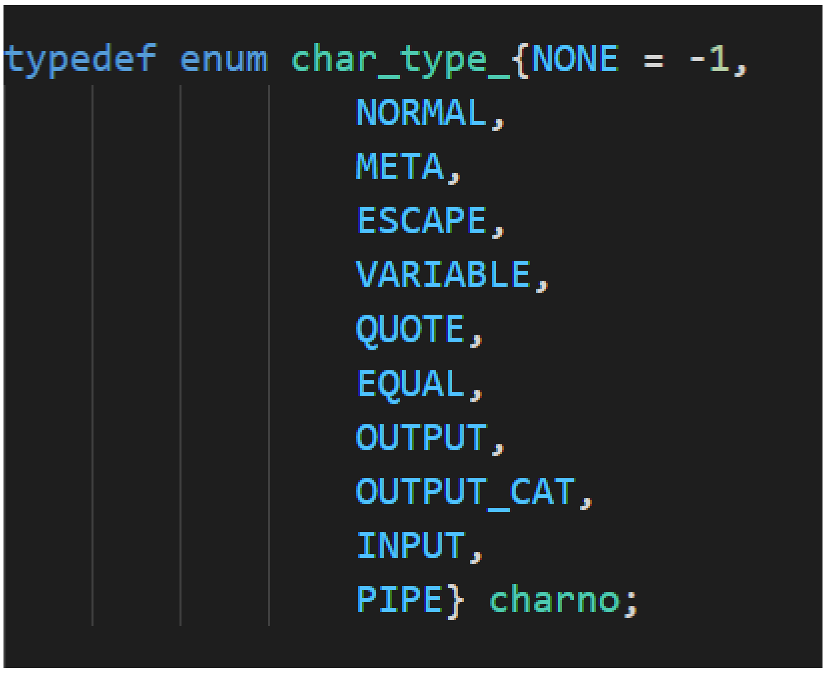
The following definitions are used to initialise the array *errors*, which holds all custom error messages, as shown below. Notice how there are 19 different error messages, but there are 20 total types of error. This is because *SYSTEM_CALL_ERROR* does not have a custom error message and *perorr()* is displayed instead, to cater for the different reasons why a system call might fail.



```c
#define ERRORS_LENGTH 19
#define MEMORY_ERROR_MSG "A problem occured while dynamically allocating memory\n"
#define BUFFER_ERROR_MSG "Did a token exceed its maximum buffer size?\n"
#define PARSE_ERROR_MSG "Error while parsing.\n"
#define VARIABLE_DECLARATION_MSG "Do you have a variable expansion character in quotes or referencing illegal characters?\n"
#define VARIABLE_EXPANSION_MSG "Variable does not exsist.\n"
#define VARIABLE_ASSIGNMENT_MSG "Invalid 'assign' operation.\n"
#define VARIABLE_NAME_MSG "An invalid variable name was provided.\n"
#define NODE_NOT_FOUND_MSG "Reference to non existent variable.\n"
#define NODE_ASSIGNMENT_MSG "Could not create new variable.\n"
#define STACK_FULL_MSG "Could not add to stack. Stack is full.\n"
#define STACK_EMPTY_MSG "Could not pop from stack. Stack has only one item.\n"
#define TOKENS_MEMORY_MSG "A problem occured while dynamically allocating memory for tokens.\n"
#define VARINDICES_MEMORY_MSG "A problem occured while dynamically allocating memory for variable pointers.\n"
#define INVALID_ARGS_MSG "Invalid arguments.\n"
#define ENV_VARIABLE_NOT_FOUND_MSG "Enviroment variable could not be found.\n"
#define ENV_VARIABLE_ASSIGNMENT_MSG "A problem occured while assigning to an enviroment variable.\n"
#define CWD_NOT_FOUND_MSG "The Current Working Directory could not be found.\n"
#define NULL_GIVEN_MSG "The program encountered a NULL instead of a value.\n"
#define NOT_A_DIR_MSG "Not a directory.\n"
```

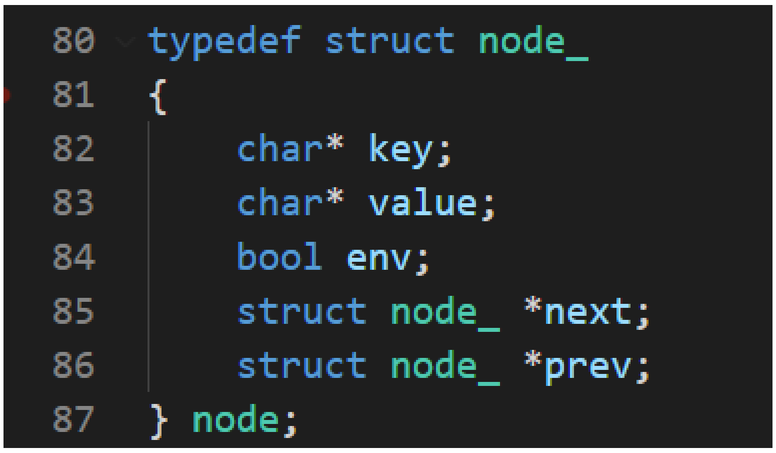
INTERNAL_COMMANDS_LEN refers to the size of the array *internal_commands.*

cmds refer to the different internal commands. They are used as an index for the array *internal_commands* .



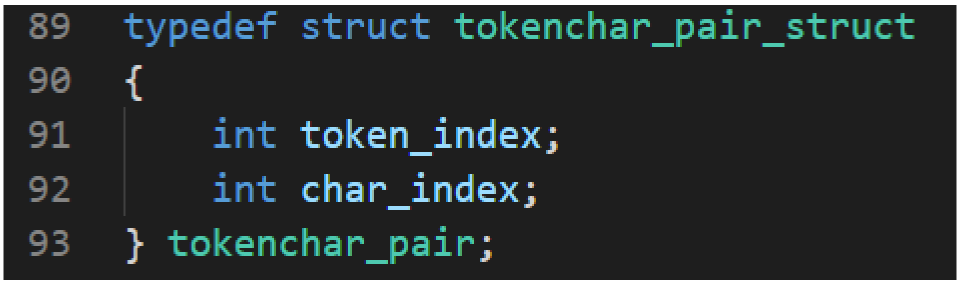
```c
typedef enum char_type_{NONE = -1,
                        NORMAL,
                        META,
                        ESCAPE,
                        VARIABLE,
                        QUOTE,
                        EQUAL,
                        OUTPUT,
                        OUTPUT_CAT,
                        INPUT,
                        PIPE} charno;
```

node refers to a singular item within the shell variable linked list. An item has a key, value, a Boolean indicating whether it is an environment variable or not, and two pointers to the previous and next nodes, if any.



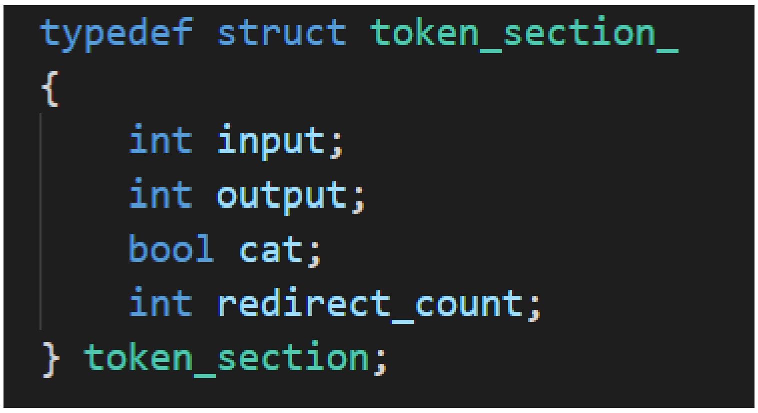
```c
typedef struct node_
{
    char* key;
    char* value;
    bool env;
    struct node_ *next;
    struct node_ *prev;
} node;
```

tokenchar_pair is used for variable expansion. *token_index* is an index to a token in *tokens* where there is the variable expansion character ($). *char_index* is the index of the variable expansion character in the token pointed to by *token_index.*



```c
typedef struct tokenchar_pair_struct
{
    int token_index;
    int char_index;
} tokenchar_pair;
```

token_section is used for redirecting external commands. An instance of this struct relates to a particular command followed by arguments and optional redirects. So when the program is executing an external command. It simply checks the values of the *token_section* corresponding to it, to determine if it should redirect to an input/output stream or not, and how many redirects are there in the command.


```
typedef struct token_section_
{
    int input;
    int output;
    bool cat;
    int redirect_count;
} token_section;
```

redirect_ext is used for redirecting external commands. Notice the array of *token_section* structs. *pipe_count* refers to the number of pipe characters present across the input string, but also to the length of *section* (+1). If *pipe_count* is 0, then it means there is only one command in the input, and hence token_section should only have one instance of a struct to determine if this command should be redirected.

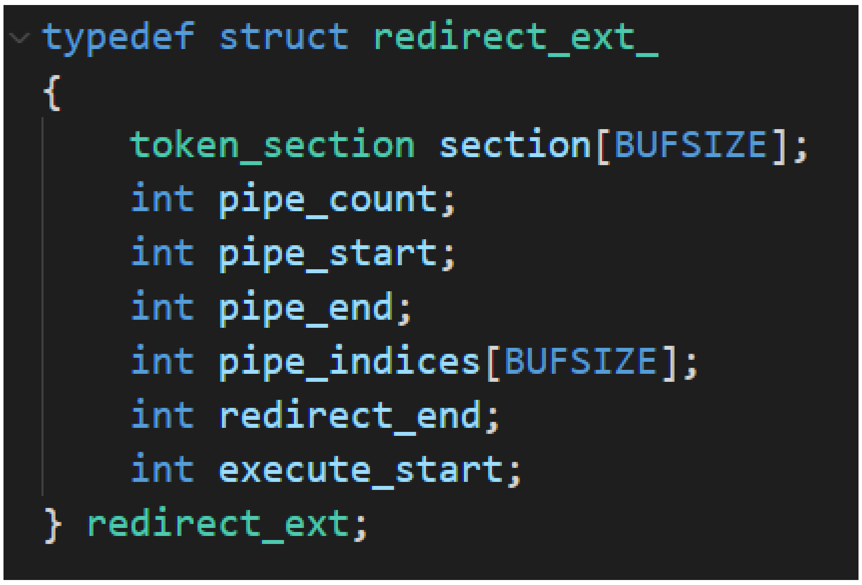
pipe_start is an index to a token in *tokens* immediately following the first pipe character encountered. This is mainly used to check whether there is a singular pipe character in the beginning of the input string, which should raise an error.

pipe_end is an index to a token in *tokens* immediately following the last pipe character encountered.

pipe_indices is an array where each element corresponds to the first token in *tokens* following a pipe character. *pipe_start* is the first element of this array and *pipe_end* is the last.

redirect_end is used in *tokens_init()*. It is an index to the current redirect character. It is called *redirect_end* because after all redirect characters are traversed, this variable will be pointing to the last redirect character.

execute_start is used when the chunks of tokens between pipes are separated. It refers to the index of a token which represents the current command to execute. See *execute_external().*



```c
typedef struct redirect_ext_
{
    token_section section[BUFSIZE];
    int pipe_count;
    int pipe_start;
    int pipe_end;
    int pipe_indices[BUFSIZE];
    int redirect_end;
    int execute_start;
} redirect_ext;
```

redirect_int is used for redirecting internal commands. 2 different systems are used for redirecting internal and external commands.

redirect_indices is an array where each element is an index to a token in *tokens* following a redirect character. This is used to determine the token in *tokens* which refers to a file name, since it is preceded by a redirect character.

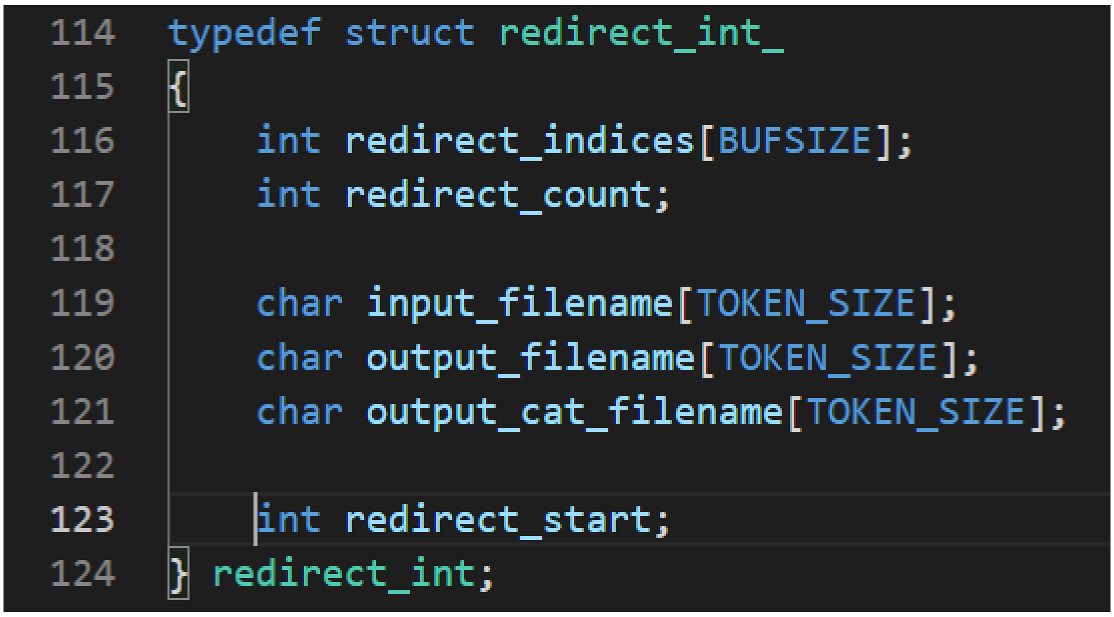
redirect_count refers to the number of redirects in the input string. It also refers to the length of *redirect_indices*.

input_filename holds the name of the file which the input should be retrieved from.

output_filename holds the name of the file which the output should be redirected to.

ouput_cat_filename holds the name of the file which the output should be appended to.

redirect_start refers to the token following the first redirect character. This is mainly used to determine if there is a redirect character at the beginning of the input string, and an error is raised if so.



```
typedef struct redirect_int_
{
    int redirect_indices[BUFSIZE];
    int redirect_count;

    char input_filename[TOKEN_SIZE];
    char output_filename[TOKEN_SIZE];
    char output_cat_filename[TOKEN_SIZE];

    int redirect_start;
} redirect_int;
```

STACK_SIZE refers to the length of the array *stack*.

child_pids is used to store the pids of the current smash session's children. The array is used when a SIGINT is sent, and all child processes are killed.

child_count refers to the length of *child_pids*.

error_msg is an array that holds all error messages.

prompt_default is a string that holds the default prompt string displayed on every new line.

metacharacters is an string where each character is considered a 'meta' character by the shell. See previous character type definitions above.

quotes is a string where each character is considered a ‘quote’ character by the shell. See previous character type definitions above.

internal_commands is an array that holds all the commands the shell can execute with no external dependencies. See previous internal commands definitions above.

error is assigned the return type of functions (See previous *err* definitions above). If this variable is non-zero, then it is used an index for *errors* in order to display the appropriate error message, in the function *handle_error()*.

exit_value refers to the value the program should return upon exiting.

exit_program refers whether the while loop encapsulating the main program should continue executing or not.

fp refers to the file pointer which the source internal command uses to take its input from.

head refers to the first value of the shell variable doubly linked list.

tail refers to the last value of the shell variable doubly linked list.

stack is the underlying array for the directory stack.

top is the index of the last item in *stack*. It is initialised to -1 if the stack is empty.

stdin_fd refers to a file descriptor which internal commands should get their input from

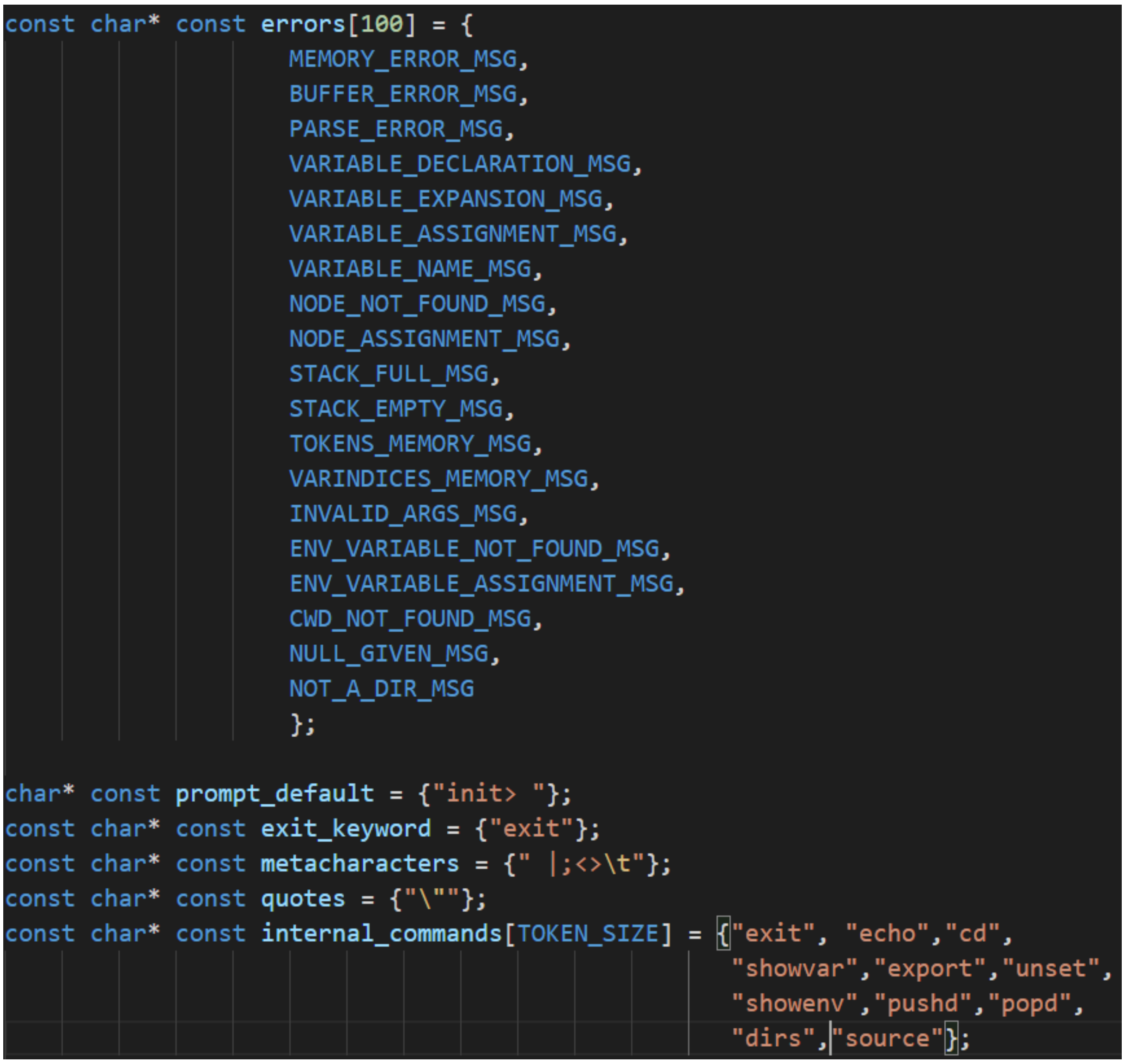
stdout_fd refers to a file descriptor which internal commands should output to.

environ is an array of all the environment variables.

All, the function protocols of *methods.c* are also listed.

Main function

First, it initialises the global variables first declared in *headers.h* to the values as shown below.



```
const char* const errors[100] = {
                    MEMORY_ERROR_MSG,
                    BUFFER_ERROR_MSG,
                    PARSE_ERROR_MSG,
                    VARIABLE_DECLARATION_MSG,
                    VARIABLE_EXPANSION_MSG,
                    VARIABLE_ASSIGNMENT_MSG,
                    VARIABLE_NAME_MSG,
                    NODE_NOT_FOUND_MSG,
                    NODE_ASSIGNMENT_MSG,
                    STACK_FULL_MSG,
                    STACK_EMPTY_MSG,
                    TOKENS_MEMORY_MSG,
                    VARINDICES_MEMORY_MSG,
                    INVALID_ARGS_MSG,
                    ENV_VARIABLE_NOT_FOUND_MSG,
                    ENV_VARIABLE_ASSIGNMENT_MSG,
                    CWD_NOT_FOUND_MSG,
                    NULL_GIVEN_MSG,
                    NOT_A_DIR_MSG
                    };

char* const prompt_default = {"init> "};
const char* const exit_keyword = {"exit"};
const char* const metacharacters = {" |;<>\t"};
const char* const quotes = {"\""};
const char* const internal_commands[TOKEN_SIZE] = {"exit", "echo","cd",
                                                    "showvar","export","unset",
                                                    "showenv","pushd","popd",
                                                    "dirs","source"};
```

The following local variables are also initialised.

input – Holds the input string

token_num – The number of tokens

tokens – A pointer to an array of individual tokens

assign_count – Number of variable assignment statements

assign_indices – A pointer to an array of *int.*

var_indices – A pointer to an array of tokenchar_pair structs.

var_indices_len – The length of the array *var_indices.*

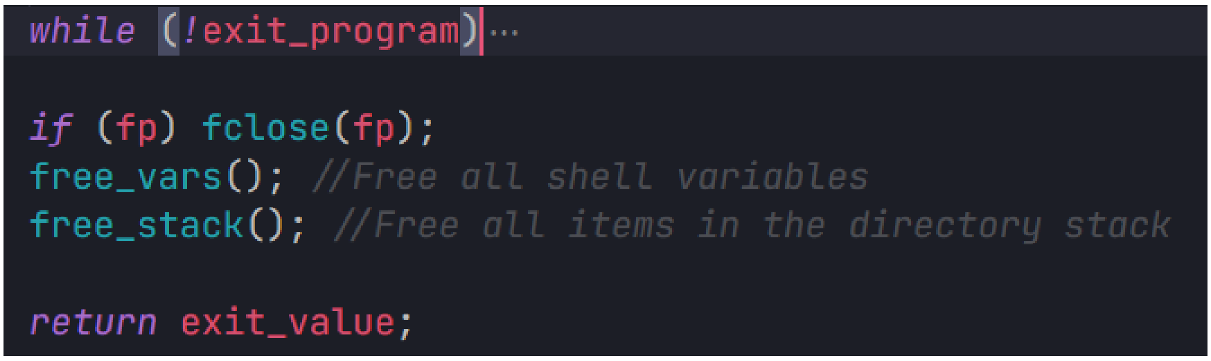
prompt – Holds the string that is displayed every time the program is expecting input from *stdin*.

ex is an instance of *redirect_ext*

in is an instance of *redirect_int*

Firstly the structs *ex* and *in* are initialised to default values by calling the *reset_ex* and *reset_in* respectively. The global variables *top, stdin_fd* and *stdout_fd* are set to -1 as default values. *top* should point to the top-most value in the diurectory stack. Since the directory stack is initially empty, it points to -1. *stdin_fd* and *stdout_fd* are used to redirect input/output streams. Initially the program is hooked to standard input/output so they are set to -1 to indicate they are not being used.

The shell and environment variables are then initialised by calling the *init_vars()* function. If a problem occurs, an appropriate error message is displayed by *handle_error().* Then the while loop will not execute and the program frees any shell variables and stack entries. *fp* is also closed if opened.

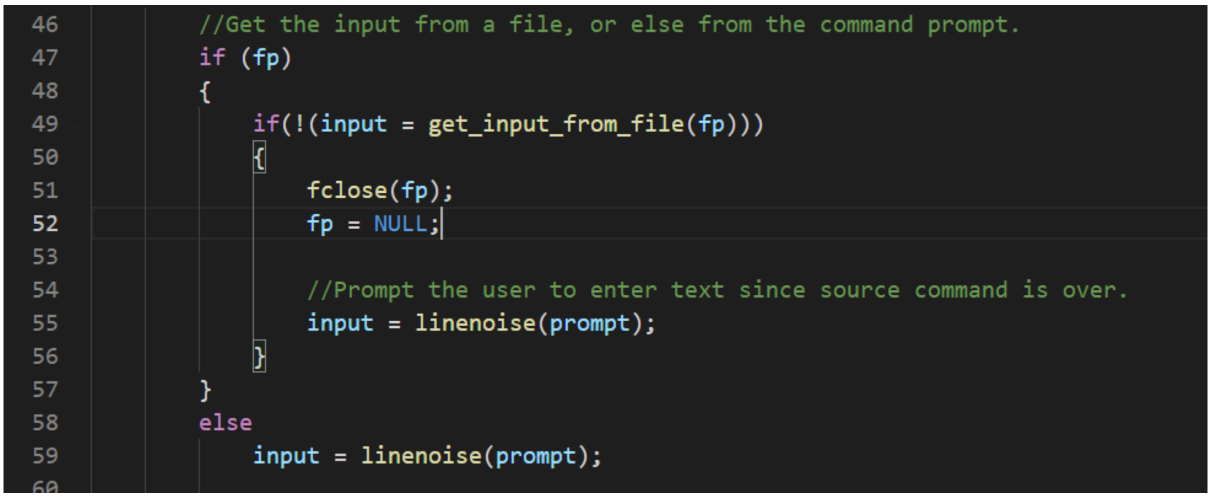


```
while (!exit_program)…

if (fp) fclose(fp);
free_vars(); //Free all shell variables
free_stack(); //Free all items in the directory stack

return exit_value;
```

The variable *prompt* is initialised to the value of the shell variable *PROMPT*, via the *node_search()* function.

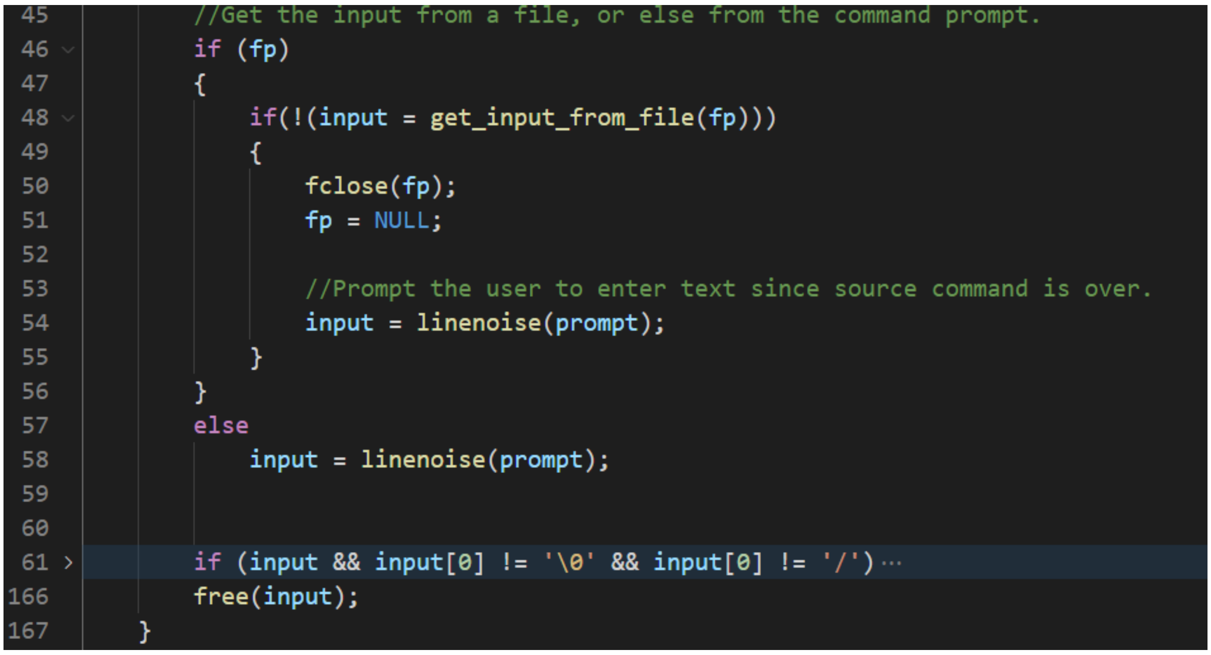
From this point onward, the program keeps repeating as long as the program is not trying to exit. The program either gets its input from a file or from *stdin*. This is done by checking if the file pointer *fp* is *NULL* or not. If it is, then the *get_input_from_file()* function reads a line from that file pointer and returns it. If *NULL* is returned, then the file pointer is closed, set to *NULL* and the input is retrieved from *stdin*.

The input is received from *stdin* via the *linenoise()* function.



```
        //Get the input from a file, or else from the command prompt.
        if (fp)
        {
            if(!(input = get_input_from_file(fp)))
            {
                fclose(fp);
                fp = NULL;

                //Prompt the user to enter text since source command is over.
                input = linenoise(prompt);
            }
        }
        else
            input = linenoise(prompt);
```

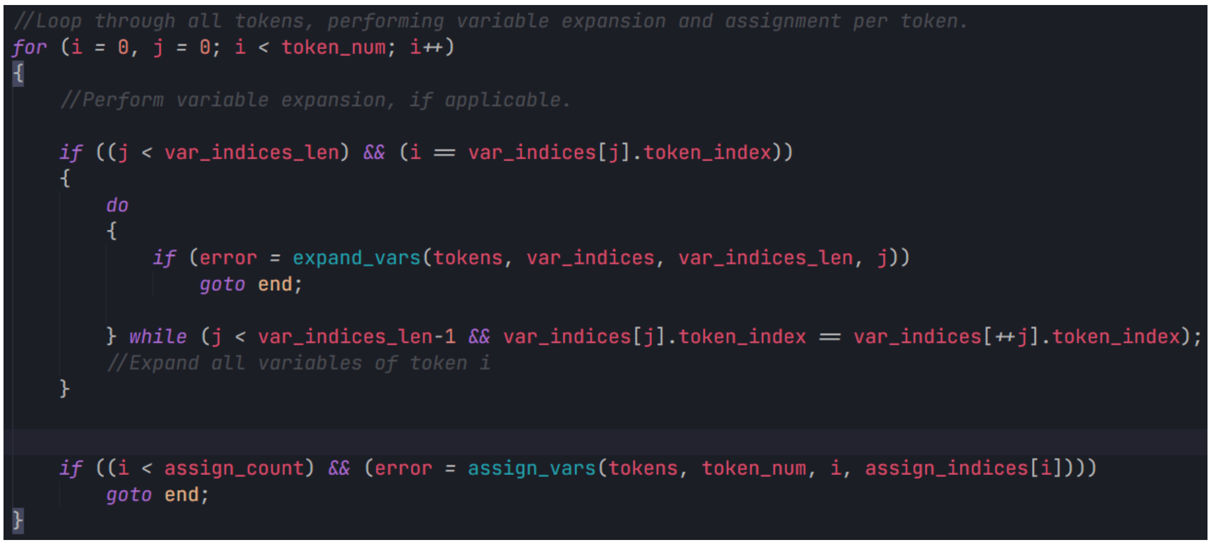
If the input received was not NULL, did not start with a *null terminator* character or a forward slash, then the program continues, otherwise it frees *input* and the next iteration is started.



```
        //Get the input from a file, or else from the command prompt.
        if (fp)
        {
            if(!(input = get_input_from_file(fp)))
            {
                fclose(fp);
                fp = NULL;

                //Prompt the user to enter text since source command is over.
                input = linenoise(prompt);
            }
        }
        else
            input = linenoise(prompt);


        if (input && input[0] != '\0' && input[0] != '/')···
        free(input);
    }
```

The input string is passed to *tokens_get(),* alongside the memory addresses of the variables *token_num, var_indices* and *var_indices_len*. If *NULL* is returned then the program jumps to the label *end*.

Then for every token in *tokens,* the program checks whether that token is representing any keys and replaces all keys within that token with their value, as shown in the *do while* loop below. Then it checks whether that token is representing a variable assignment operation and if so calls *assign_vars* with the index of the variable assignment statement.



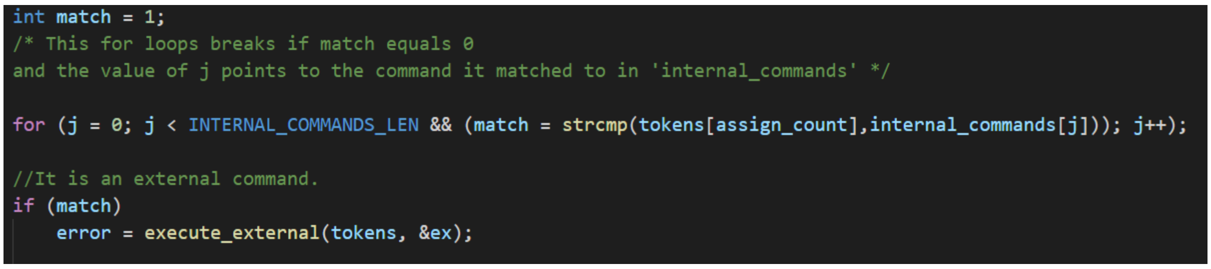
```c
//Loop through all tokens, performing variable expansion and assignment per token.
for (i = 0, j = 0; i < token_num; i++)
{
    //Perform variable expansion, if applicable.

    if ((j < var_indices_len) && (i == var_indices[j].token_index))
    {
        do
        {
            if (error = expand_vars(tokens, var_indices, var_indices_len, j))
                goto end;

        } while (j < var_indices_len-1 && var_indices[j].token_index == var_indices[++j].token_index);
        //Expand all variables of token i
    }


    if ((i < assign_count) && (error = assign_vars(tokens, token_num, i, assign_indices[i])))
        goto end;
}
```

The variable j refers to the index of the next *struct* in *var_indices*. Since *i* refers to the current token, and var_indices[j]->token_index, refers to the index of the token in *tokens* where there is supposed to be a variable to be expanded, then if *i* == *var_indices[j]->token_index,* then it means the token pointed to by *i* is a variable that needs to be expanded. Moreover, as long as *j* is less than *var_indices_len* it means that there are variables in *tokens* that still need to be expanded. Once these two conditions are met, the function *expand_vars()* replaces the key in *tokens*, with its variable's value. If *expand_vars()* return a non-zero number, it means an error has occurred, and the program jumps to the label *end*.

If *assign_count* is less than the total number of tokens, then that means there is at least one command to execute, otherwise the program does not execute the logic to execute an internal/external command because the whole token space was just a series of variable assignment statements. This is evident by the following if statement.

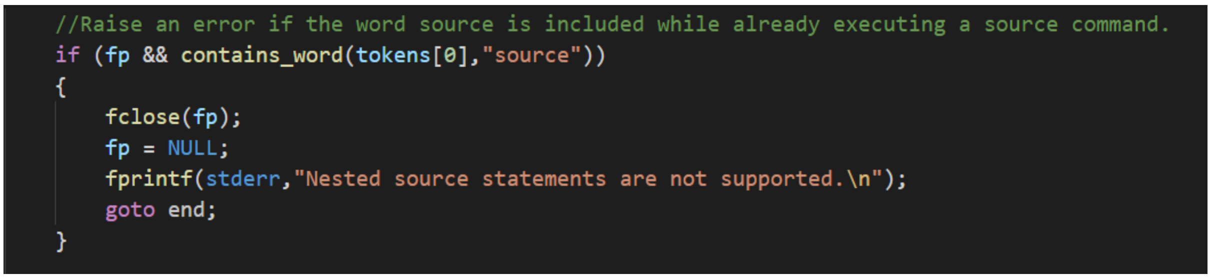
If there was the pipe character in the input string, then the tokens are interpreted as external commands, since piping is not supported for internal commands. This is done by checking if the variable *pipe_count* in the *struct ex* is not equal to 0. The function *execute_external* will look for a binary image in the *PATH* environment variable and execute it.

Otherwise, the token at *assign_count* is compared with every element in the array *internal_commands.* If there is a match, then *execute_internal()* is run. This is achieved by checking whether *match* is 0 or not. If *match* is non zero, it means that the first token did not match with an element in *internal_commands* and hence it is an external command. The token at *assign_count* is chosen because *assign_count* is 0 if there are no preceding variable assignments, meaning the first token is the first command, otherwise the first command is the token immediately after the variable assignment statements.



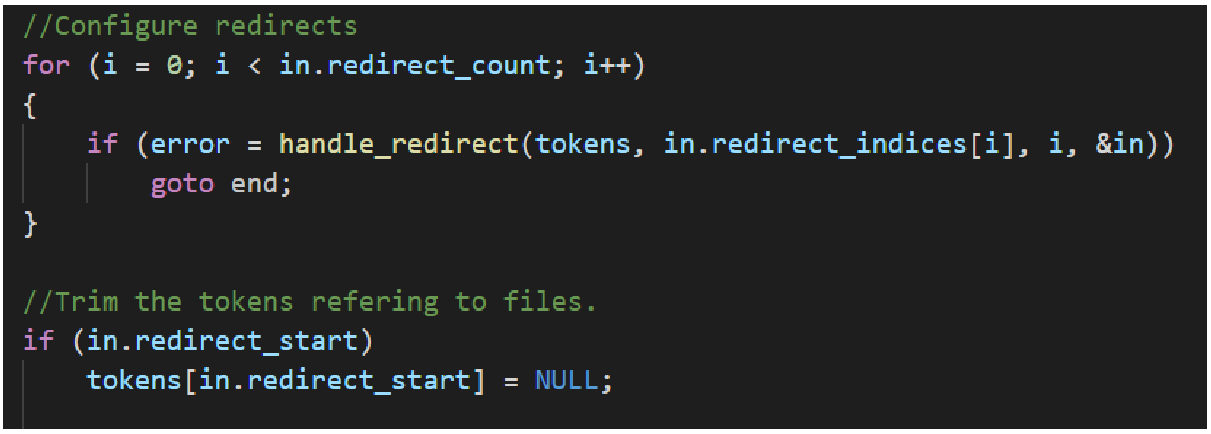
```c
int match = 1;
/* This for loops breaks if match equals 0
and the value of j points to the command it matched to in 'internal_commands' */

for (j = 0; j < INTERNAL_COMMANDS_LEN && (match = strcmp(tokens[assign_count],internal_commands[j])); j++);

//It is an external command.
if (match)
    error = execute_external(tokens, &ex);
```

If *match* was 0, then it means the first command is an internal command. If the input is being read from a file, check if the first token is *source* or not. Nested source statements are not allowed, so an error is promptly raised and the file pointer *fp* is closed. The program then jumps to the *end* label.



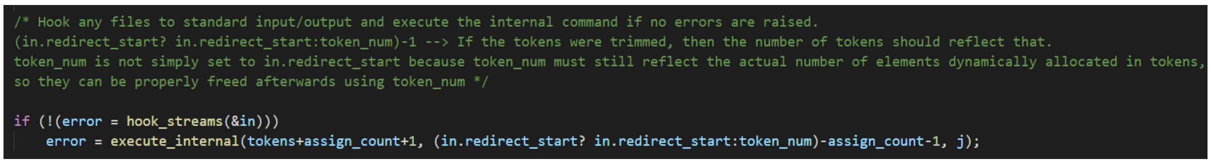
```c
    //Raise an error if the word source is included while already executing a source command.
    if (fp && contains_word(tokens[0],"source"))
    {
        fclose(fp);
        fp = NULL;
        fprintf(stderr,"Nested source statements are not supported.\n");
        goto end;
    }
```

The program then checks whether the tokens are redirecting the input or output to a file. The *redirect_indices* in *struct in* holds the index of the tokens that are referring to a file. As a result, after the redirects are handled through the *handle_redirect()* function, the tokens are spliced such that the tokens referring to file names are removed. This is so the tokens referring to file names are not treated as arguments when executing the internal command.



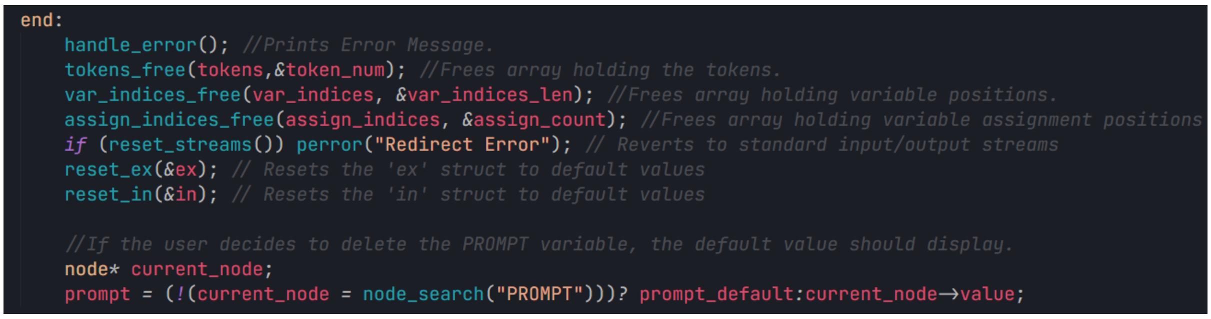
```c
//Configure redirects
for (i = 0; i < in.redirect_count; i++)
{
    if (error = handle_redirect(tokens, in.redirect_indices[i], i, &in))
        goto end;
}

//Trim the tokens refering to files.
if (in.redirect_start)
    tokens[in.redirect_start] = NULL;
```

Then the streams are hooked to their respective file through the *hook_streams()* function. If no errors occurred then *execute_internal()* is called. If any tokens were removed from *tokens* then *in.redirect_start-1* is passed as the number of arguments. Otherwise *token_num-1* is passed.



```c
/* Hook any files to standard input/output and execute the internal command if no errors are raised.
(in.redirect_start? in.redirect_start:token_num)-1 --> If the tokens were trimmed, then the number of tokens should reflect that.
token_num is not simply set to in.redirect_start because token_num must still reflect the actual number of elements dynamically allocated in tokens,
so they can be properly freed afterwards using token_num */

if (!(error = hook_streams(&in)))
    error = execute_internal(tokens+assign_count+1, (in.redirect_start? in.redirect_start:token_num)-assign_count-1, j);
```

Finally the program frees all arrays, and resets any variables to default values so they may be ready for the next iteration. This is also where the label *end* is. This is because if an error occurred, the program must clean up before the next iteration. *handle_error(), tokens_free(), var_indices_free(), reset_streams(), reset_ex()* and *reset_in()* are run. Moreover, the program checks whether the shell variable *PROMPT* was deleted or not. If so, *prompt* is set to *prompt_default*, otherwise it is set to the value of the shell variable PROMPT.



```
end:
    handle_error(); //Prints Error Message.
    tokens_free(tokens,&token_num); //Frees array holding the tokens.
    var_indices_free(var_indices, &var_indices_len); //Frees array holding variable positions.
    assign_indices_free(assign_indices, &assign_count); //Frees array holding variable assignment positions
    if (reset_streams()) perror("Redirect Error"); // Reverts to standard input/output streams
    reset_ex(&ex); // Resets the 'ex' struct to default values
    reset_in(&in); // Resets the 'in' struct to default values

    //If the user decides to delete the PROMPT variable, the default value should display.
    node* current_node;
    prompt = (!(current_node = node_search("PROMPT")))? prompt_default:current_node->value;
```

Finally, *input* is freed.

Methods Used

Shell variable doubly Linked List

The terms *node* and *item* are used interchangeably.

node_insert

Function protocol:

err node_insert(const char* const key, const char* const value, bool env);

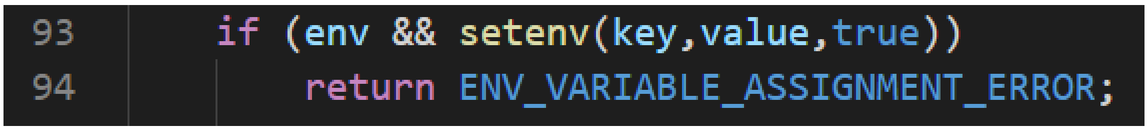
This function adds a new item to the shell variable doubly linked list.

key is the key of the new item.

value is the value of the new item.

env is the value of the *env* Boolean variable of the new item.

If a node with the same key already exists, it is deleted and the function continues. It dynamically allocates memory to the new item, including its *key* and *value* variables, and the parameters passed are used to fill the item. If *env* is set to true, then a new environment variable with the *key* and *value* provided is created using the *setenv()* function. Finally a new node with the parameters specified is added *as the head* to the shell variable doubly linked list and any other nodes have their pointers appropriately modified to handle this new change.



```
    if (env && setenv(key,value,true))
        return ENV_VARIABLE_ASSIGNMENT_ERROR;
```

The function returns 0 on success and an *err* instance on failure.

node_search

Function protocol: node* node_search(const char* key);

This function searches for a node in shell variable doubly linked list.

key is the key of the item to search for

This function traverses the linked list comparing *key* with the current node's key.

If the linked list is empty, NULL is returned, and *error* is set to indicate the error. Otherwise, the pointer to the first node with the same key values is returned.

node_delete

Function protocol: err node_delete(node* current_node);

This function deletes the node passed in from the shell variable doubly linked list. If that variable is also an environment variable then it is deleted from the environment using *unsetenv()*.

current_node is a pointer to the node to be deleted

The item is removed from the linked list and the previous and next items are joined together. All dynamically allocated variables in that item are freed and then the pointer to that item is freed too.

If *current_node* is NULL, *NODE_NOT_FOUND_ERROR* is returned. It is the programmer's responsibility to pass in a valid pointer to a node. Otherwise the behaviour is undefined.

The function returns 0 on success and an *err* instance on failure.

node_edit

Function protocol: err node_edit(node* current_node, const char* value);

This function changes the node provided's *value* variable to the parameter *value*.

The function reallocates memory to *current_node*'s *value*, to the length of the parameter *value*. Then *current_node*'s *value* is updated. If *current_node*'s *env* was true, then *setenv()* is used to update the environment variable corresponding to it.

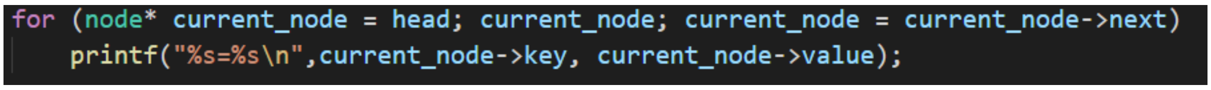
The function returns 0 on success and an *err* instance on failure.

nodes_print

Function protocol: void nodes_print()

This function prints all key value pairs of the shell variable doubly linked list

This is achieved through a for loop.



```
for (node* current_node = head; current_node; current_node = current_node->next)
    printf("%s=%s\n",current_node->key, current_node->value);
```

The function does not return anything.

node_export

Function protocol: err node_export(node* current_node);

This function takes a pointer to an existing shell variable and sets its *env* variable to *true*. Then the environment variable corresponding to that variable is created (or updated) with the new value. It is the programmer's responsibility to pass in a valid *node* pointer. Otherwise the behaviour is undefined.

The function returns 0 on success and an *err* instance on failure.

Stack Directory

Note: The following functions do not check if the directory stack is empty or not before performing an operation. This is because the directory stack will have one value pushed into it in the *init_vars()* function. Moreover, *pop()* cannot be used to remove the only item in *stack*. Therefore *stack* will never be empty.

is_full

Function protocol: bool is_full()

This function checks if the number of elements in *stack* is equal to *STACK_SIZE.*

The function return true on success and false on failure.

push

Function protocol: err push(const char* value);

This function adds a new item to *stack*.

value is the string to be added to the *stack*.

It dynamically allocates a character pointer in the next available entry in *stack*, and *value* is added to that position. *top* is incremented to reflect the new changes, and the current working directory is changed via the *change_directory* function. This is because the top most element of *stack* must always reflect the current working directory of the program.

The function returns 0 on success and an *err* instance on failure.

pop

Function protocol: err pop(char** value);

This function removes the top most element in *stack*.

value is the memory address which will point to a copy of the popped item.

value is dynamically assigned memory. The function copies the top most item into the memory address pointed to by *value*, decrements *top* to reflect the new changes, frees the popped item, and changes the current working directory via the *change_directory()* function. This is because the top most value of the stack must always reflect the current working directory.

If the stack only has one item, the function exits early and returns a positive integer.

The function returns 0 on success and an *err* instance on failure.

peek

Function protocol: err peek(char** value);

This function copies the top most item into the memory location pointed to by *value.*

value is the memory address which will point to a copy of the top most item.

value is dynamically assigned memory. This function copies the top most item into the memory address pointed to by *value*.

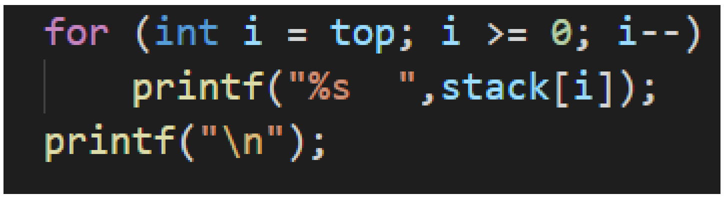
If *stack* only has one item, the function exits early and returns an *err* instance.

The function returns 0 on success and an *err* instance on failure.

print_stack

Function protocol: err print_stack();

This function prints all the items in *stack*, starting from the top most value. A new line character is printed afterwards. This is achieved through a for loop as shown below.



```
for (int i = top; i >= 0; i--)
    printf("%s  ",stack[i]);
printf("\n");
```

This function returns 0.

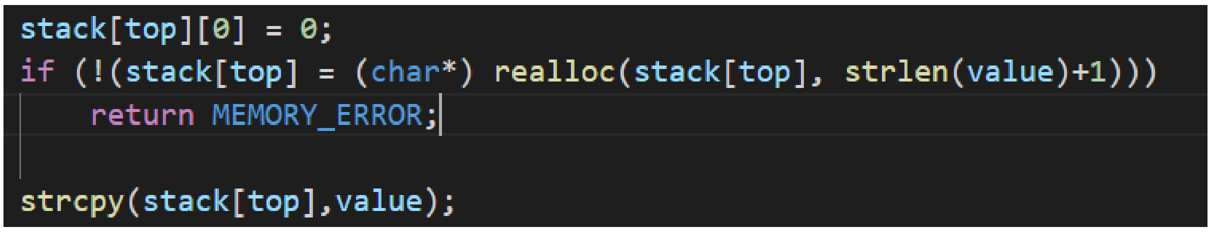
change_topmost

Function protocol: err change_topmost(const char* value);

This function changes the top most item of *stack* to *value*.

value is the string which the top most item of *stack* will be changed to.

This function simply reallocates memory to the top most item of *stack* to fit the length of *value*. Then, *value* is copied to it.



```
stack[top][0] = 0;
if (!(stack[top] = (char*) realloc(stack[top], strlen(value)+1)))
    return MEMORY_ERROR;

strcpy(stack[top],value);
```

The function returns 0 on success and an *err* instance on failure.

change_directory

Function protocol: err change_directory(const char* cwd);

This function changes the top most value of *stack*. It also updates the environment variables *PWD* and *CWD*, and the shell variable *CWD*. With the value of the new directory.

cwd is a string representing the new directory.

The function uses *chdir()* to change the current working directory. Then PWD(environment) and CWD (shell and environment) are updated with the new value. Then the top most item in *stack* is updated via the *change_topmost()* function to reflect the new changes.

The function returns 0 on success and an *err* instance on failure.

Tokenisation

tokens_init

Function protocol: int tokens_init(const char* string, redirect_int* in, redirect_ext* ex);

This function checks if *string* is parse-able. This function uses *char_type()* to determine the type of the character. The conditions *string* must meet are:

- No Redirect or Pipe Characters without a preceding token.
- No Redirect or Pipe Characters without a proceeding token.
- No Redirect or Pipe Characters not separated by a token.
- Number of Redirect Characters must not exceed BUFSIZE (as defined in *headers.h*).

- Number of Pipe Characters must not exceed BUFSIZE.
- Only the following Redirect Characters are supported. (<) (>) (>>)
- Even number of Quote Characters (“).
- No Quote Characters directly adjacent to each other.
- Variable Character must be proceeded by a Normal Character or a Quote Character
- Last character cannot be an Escape Character or a Variable Character
- String must have at least one Normal Character.
- No Variable Characters in a quotes string of text.
- The number of Pipe Characters (ie. *ex->pipe_count*) and the number of Redirect Characters (ie. *in->redirect_count*) cannot exceed *BUFSIZE* characters.
- Only one Equal Character per token. Other instances of (=) are treated as normal
- If a token has no Equal Characters, then any other subsequent tokens will have the (=) character treated as normal (A=1 echo B=2 → outputs *B=2* and set A to 2. B is not created)

This function also fills the structs *ex* and *in* with the appropriate values in order to be used later on in *execute_internal()* and *execute_external()*.

This function returns the length *tokens* should be on success. Otherwise 0 is returned to indicate that one of the above conditions has been violated.

char_type

Function protocol: charno char_type(const char* string, int j);

This function determines the type of the character at index *j* in *string*.

j is the index of the character

string is the string with the character whose type is to be determined.

This function uses the *is_deref()* function to determine if the character at index *j* in string is dereferenced using the escape character (\).

To check if it is a meta charcter, the function *is_meta()* is used, which also uses the *is_deref()* functio.

The function returns the character type (see character type definitions) of the character, or NONE if *j* is negative.

is_meta

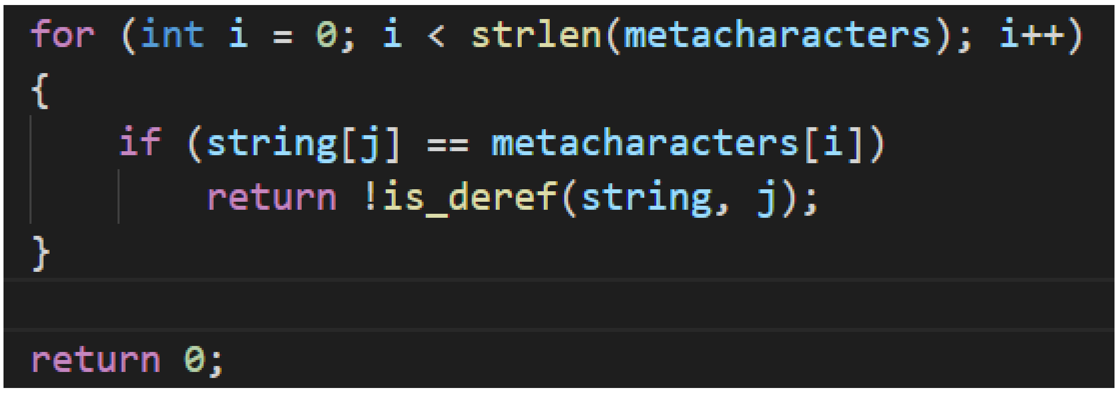
Function protocol: bool is_meta(const char* string, int j);

This function determines whether the character at index *j* in *string* is a meta character or not.

j is the index of the character in *string*.

string is the string with the character whose type is to be determined.

It loops over *metacharacters*, and if the character at index *j* in *string* matches with a character in *metacharacters* and it is not dereferenced by an escape character (this is done by using the function *is_deref()*), then true is returned, otherwise false.



```
for (int i = 0; i < strlen(metacharacters); i++)
{
    if (string[j] == metacharacters[i])
        return !is_deref(string, j);
}

return 0;
```

is_deref

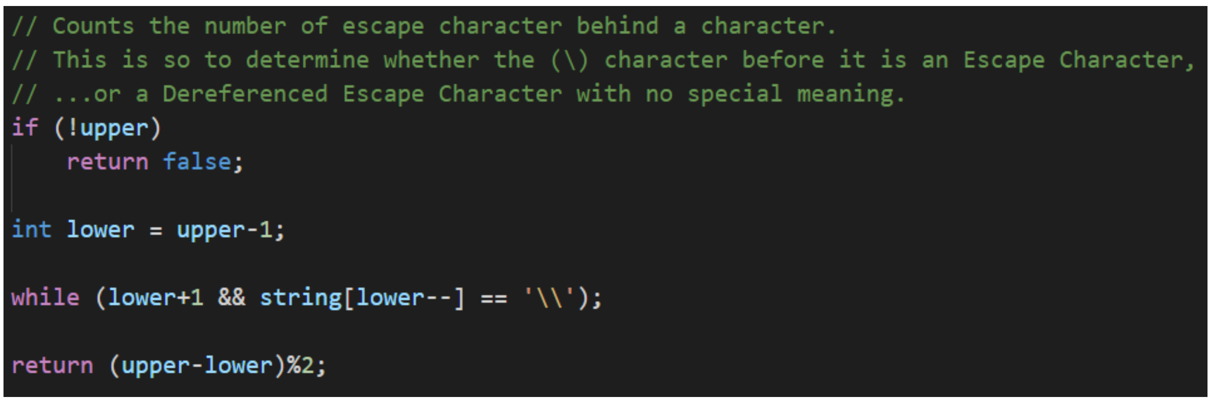
Function protocol: bool is_deref(const char* string, int upper);

This function checks whether the character at index *upper* in *string* is dereferenced by an escape character (\) before it.

upper is the index of the character in *string*.

string is the string with the character to check if its dereferenced or not.

The function counts the number of Escape Characters directly preceding the character at index *upper*. If that number is odd then true is returned (meaning the character is dereferenced), otherwise, if it is even then false is returned. If *upper* is 0, meaning it is pointing to the first character, we know that it is surely not dereferenced, since an Escape Character cannot precede it.



```
// Counts the number of escape character behind a character.
// This is so to determine whether the (\) character before it is an Escape Character,
// ...or a Dereferenced Escape Character with no special meaning.
if (!upper)
    return false;

int lower = upper-1;

while (lower+1 && string[lower--] == '\\');

return (upper-lower)%2;
```

Eg:

- \\\$ → $ is dereferenced.
- \\\\$ → $ is not dereferenced, because the (\) behind it does not carry any special meaning because that (\) is being dereferenced by the (\) behind it.

tokens_get

Function protocol:

char** tokens_get(const char* input, int* length, tokenchar_pair** var_indices, int* var_length, int** assign_indices, int* assign_count, redirect_int* in, redirect_ext* ex)

This function parses *input* into tokens. It also fills the values pointed to by the memory locations in *length*, *var_indices* and *var_index*.

input is a string which will be parsed and tokenised.

length is the memory location that will contain the number of strings in the return value upon exiting the function.

var_indices is a memory location that will contain an array which will hold a *tokenchar_pair* struct for every Variable Expansion Character encountered while parsing *input,* upon exiting the function.

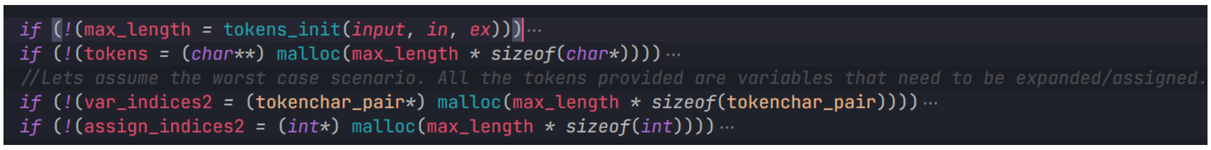
var_length is the memory location that will contain the number of *tokenchar_pair* structs in the array pointed to by *var_indices*.

assign_indices is a memory location that will contain an array of integers for every Variable Assignment Character encountered while parsing *input,* upon exiting the function.

assign_count is the memory location that will contain the number of elements in assign_indices.

For the sake of this explanation, let's refer to the return value as *tokens*, which is an array of dynamically allocated strings that will be dynamically allocated by this function. *tokens*, defined in *smash.c* holds the return value of this function, hence why they share the same name.

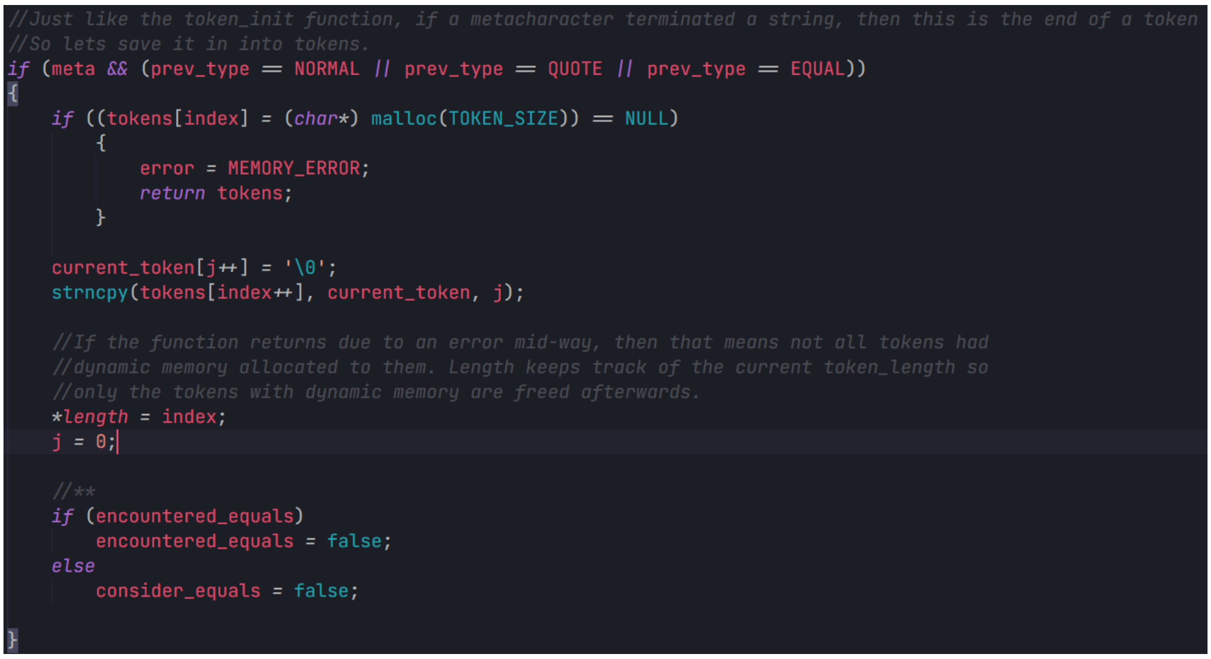
The function calls *tokens_init()*, with the struct pointers *in* and *ex* to initialise the redirects and pipes, check for any parse errors, and also receives the number of string pointers the *tokens* should be dynamically allocated. *var_indices* also uses the return value of *tokens_init()* to dynamically allocate memory.



```
if (!(max_length = tokens_init(input, in, ex)))···
if (!(tokens = (char**) malloc(max_length * sizeof(char*))))···
//Lets assume the worst case scenario. All the tokens provided are variables that need to be expanded/assigned.
if (!(var_indices2 = (tokenchar_pair*) malloc(max_length * sizeof(tokenchar_pair))))···
if (!(assign_indices2 = (int*) malloc(max_length * sizeof(int))))···
```

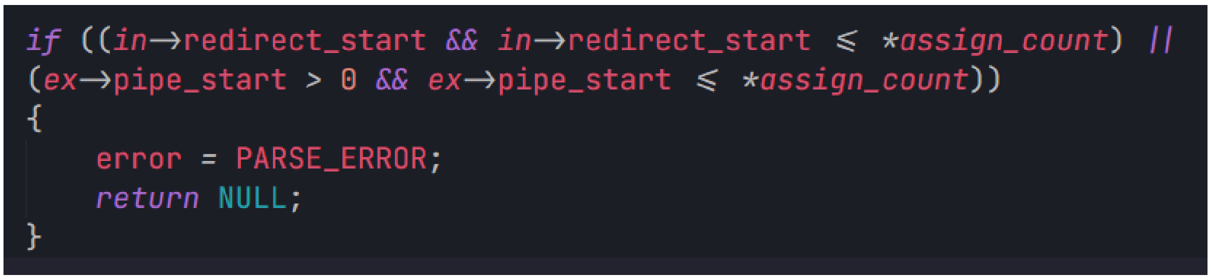
This function works with the type of the current and previous characters to determine the next course of action by using the *char_type()* function.

All Normal Characters encountered are saved in a buffer and when the function deems that a token has ended, (encountering a meta character) then that series of character is placed in *tokens*, and is allocated memory, as shown below.



```
//Just like the token_init function, if a metacharacter terminated a string, then this is the end of a token
//So lets save it in into tokens.
if (meta && (prev_type == NORMAL || prev_type == QUOTE || prev_type == EQUAL))
{
    if ((tokens[index] = (char*) malloc(TOKEN_SIZE)) == NULL)
        {
            error = MEMORY_ERROR;
            return tokens;
        }

    current_token[j++] = '\0';
    strncpy(tokens[index++], current_token, j);

    //If the function returns due to an error mid-way, then that means not all tokens had
    //dynamic memory allocated to them. Length keeps track of the current token_length so
    //only the tokens with dynamic memory are freed afterwards.
    *length = index;
    j = 0;

    //**
    if (encountered_equals)
        encountered_equals = false;
    else
        consider_equals = false;

}
```

Whenever a variable expansion character is encountered it is saved in the memory location pointed to by *var_indices*. Similiarly for *assign_indices*. If there are redirects/pipes joining at least one variable assignment statement, an error is promptly raised.



```c
if ((in->redirect_start && in->redirect_start <= *assign_count) ||
(ex->pipe_start > 0 && ex->pipe_start <= *assign_count))
{
    error = PARSE_ERROR;
    return NULL;
}
```

The function returns NULL on failure and the global variable *error* is set to indicate the error and on success a dynamically allocated null terminated array of pointers to dynamically allocated strings (*tokens*) is returned.

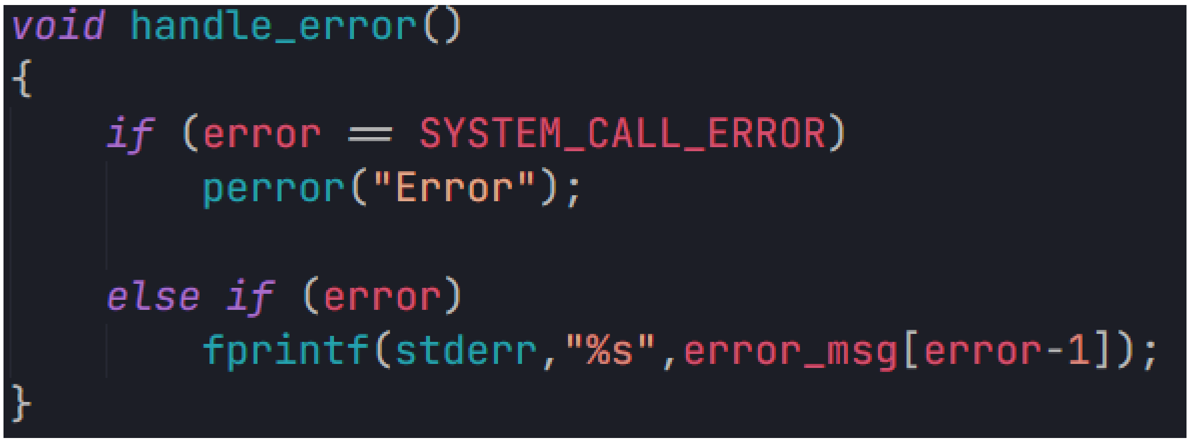
Error Handling & Variable Resetting

handle_error

Function protocol: void handle_error()

This function uses *error* as an index for the array *errors* in order to display the value of *error*'s custom error message to *stderr*.

If *error* is a *SYSTEM_CALL_ERROR*, then *perror* is used to display the error message.



```c
void handle_error()
{
    if (error == SYSTEM_CALL_ERROR)
        perror("Error");

    else if (error)
        fprintf(stderr,"%s",error_msg[error-1]);
}
```

Note that *error-1* is used an index, this is because the first value in *errors* (so index 0), corresponds to the value of *error* with value 1.

tokens_free

Function protocol: void tokens_free(char** tokens, int* length)

The number of pointers in *tokens* freed is equal to the value in *length+1.* Then *tokens* itself is freed.

length is the memory location containing the number of pointers+1 (starting from 0) to be freed from *tokens*.

tokens is a double character pointer, whose Strings will be freed from.

The function freed *length* pointers from *tokens*. After it completely frees *tokens* and its contents, the value pointed to by the memory location in *length*, is set to -1 and *tokens* is set to NULL, to indicate that *tokens* has been freed.

If *tokens* is NULL then the function does not do anything and exits normally.

var_indices_free

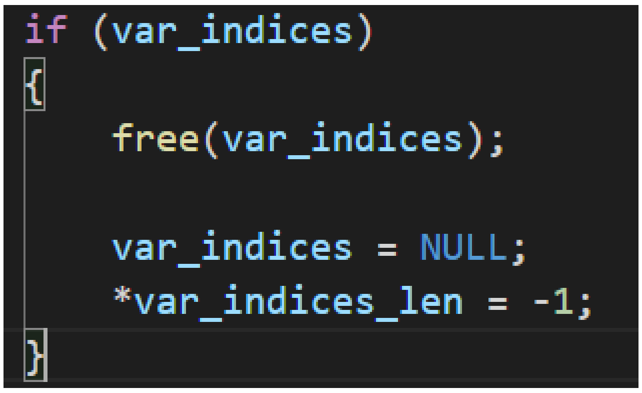
Function Protocol: void var_indices_free(tokenchar_pair* var_indices, int* var_indices_len)

var_indices_len is a memory location holding the length of *var_indices*.

var_indices is an array of *tokenchar_pair* structs which is to be freed.

If *var_indices* is NULL, then *var_indices* is not freed.

If *var_indices* was not NULL then *var_indices* is set to NULL and *var_indices_len* is set to -1.



```
if (var_indices)
{
    free(var_indices);

    var_indices = NULL;
    *var_indices_len = -1;
}
```

assign_indices_free

Function Protocol: void assign_indices_free(int* assign_indices, int* assign_count)

assign_count is a memory location holding the length of *var_indices*.

assign_indices is an array of *int* which is to be freed.

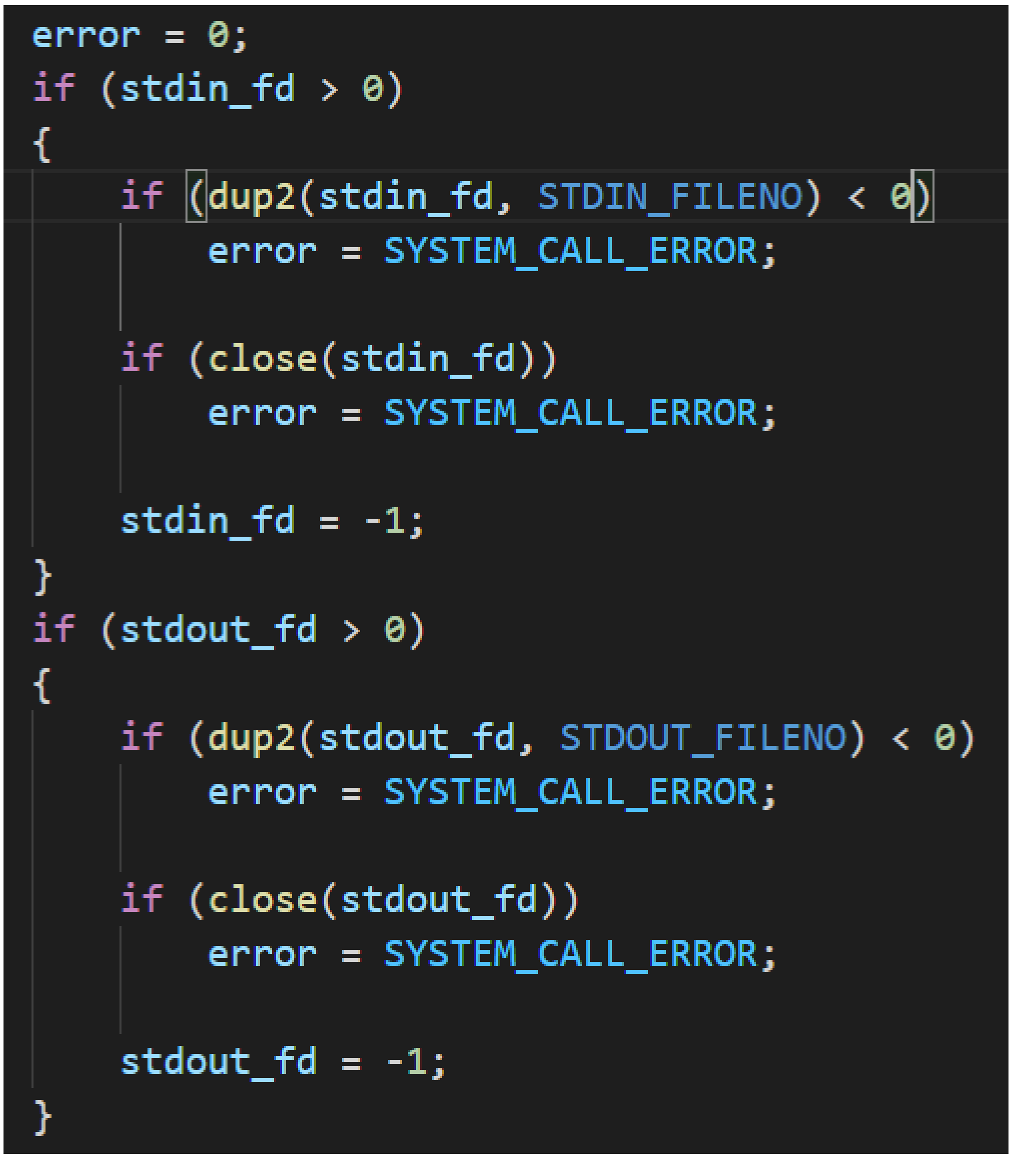
If *assign_indices* is NULL, then assign_*indices* is not freed.

If *assign_indices* was not NULL then *assign_indices* is set to NULL and *var_indices_len* is set to -1.

reset_streams

Function Protocol: err reset_streams()

This function uses the file descriptors *stdin_fd,* and *stdout_fd* to STDIN_FILENO and STDOUT_FILENO (as defined in stdio.h), respectively by using the *dup2()* function. It then sets them to -1 to indicate they have been closed.



```
error = 0;
if (stdin_fd > 0)
{
    if (dup2(stdin_fd, STDIN_FILENO) < 0)
        error = SYSTEM_CALL_ERROR;

    if (close(stdin_fd))
        error = SYSTEM_CALL_ERROR;

    stdin_fd = -1;
}
if (stdout_fd > 0)
{
    if (dup2(stdout_fd, STDOUT_FILENO) < 0)
        error = SYSTEM_CALL_ERROR;

    if (close(stdout_fd))
        error = SYSTEM_CALL_ERROR;

    stdout_fd = -1;
}
```

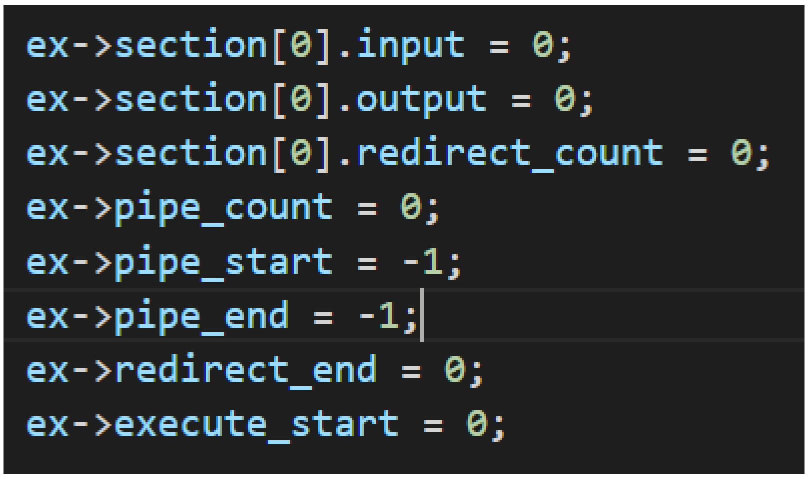
The function returns 0 on success and an *err* instance on failure.

reset_ex

Function protocol: void reset_ex(redirect_ext* ex)

This function resets the struct *ex*'s values to default values so it can be used for the next iteration.

The values set are shown below.



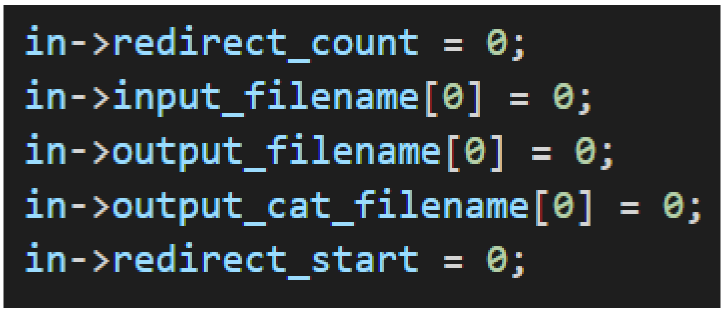
```
ex->section[0].input = 0;
ex->section[0].output = 0;
ex->section[0].redirect_count = 0;
ex->pipe_count = 0;
ex->pipe_start = -1;
ex->pipe_end = -1;
ex->redirect_end = 0;
ex->execute_start = 0;
```

reset_in

Function protocol: void reset_in(redirect_int* in)

This function resets the struct *in*'s values to default values so it can be used for the next iteration.

The values set are shown below:

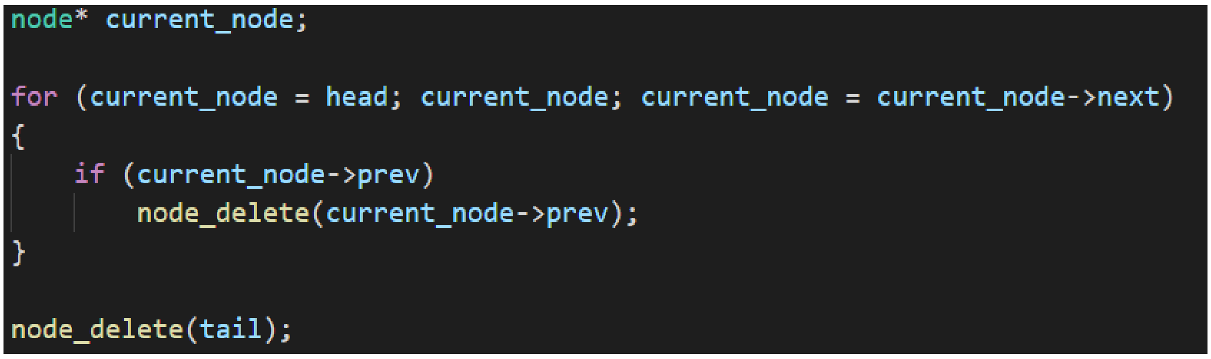


```
in->redirect_count = 0;
in->input_filename[0] = 0;
in->output_filename[0] = 0;
in->output_cat_filename[0] = 0;
in->redirect_start = 0;
```

free_vars

Function protocol: void free_vars()

This function frees all nodes in the doubly linked list. This is done by iterating over every node and deleting it with the *node_delete()* function.

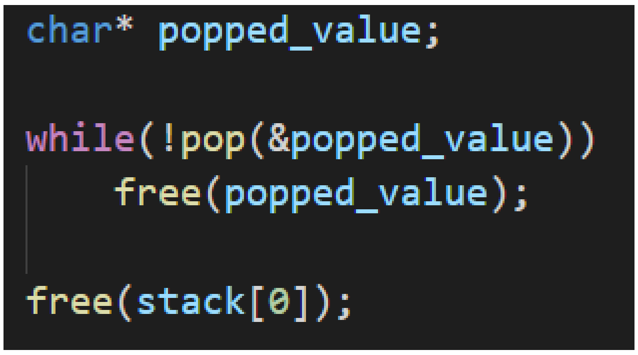


```
node* current_node;

for (current_node = head; current_node; current_node = current_node->next)
{
    if (current_node->prev)
        node_delete(current_node->prev);
}

node_delete(tail);
```

free_stack

Function protocol: void free_stack()

This function frees every value in the directory stack. This is done by using the *pop()* function and then freeing that popped variable. Lastly the last item in the stack is freed, since this cannot be freed using the *pop()* function since then the stack would be empty.



```
char* popped_value;

while(!pop(&popped_value))
    free(popped_value);

free(stack[0]);
```

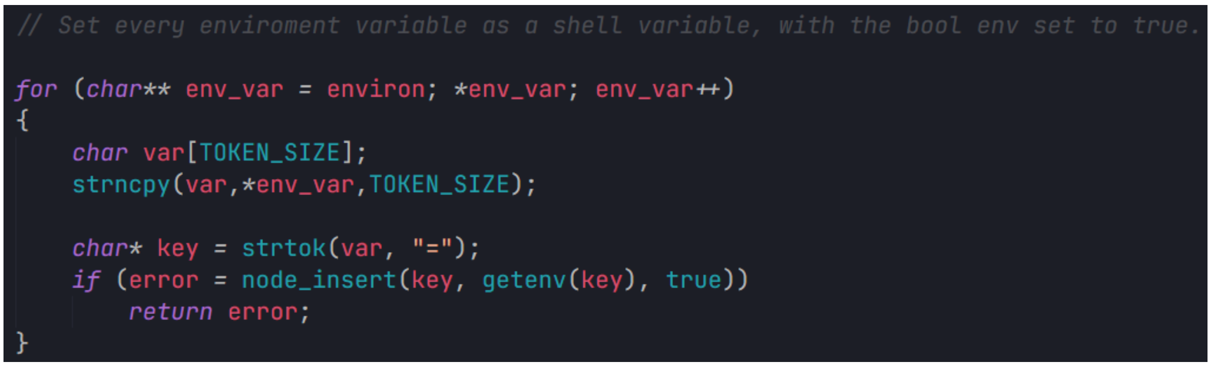
Shell variables

init_vars

Function protocol: err init_vars(void)

This function fills the shell variable doubly linked list with all the environment variables.

All the environment variables are placed in the linked list via the *node_insert* function, with the *env* argument set to true. All environment variables' value is truncated to a maximum of *TOKEN_SIZE* bytes.

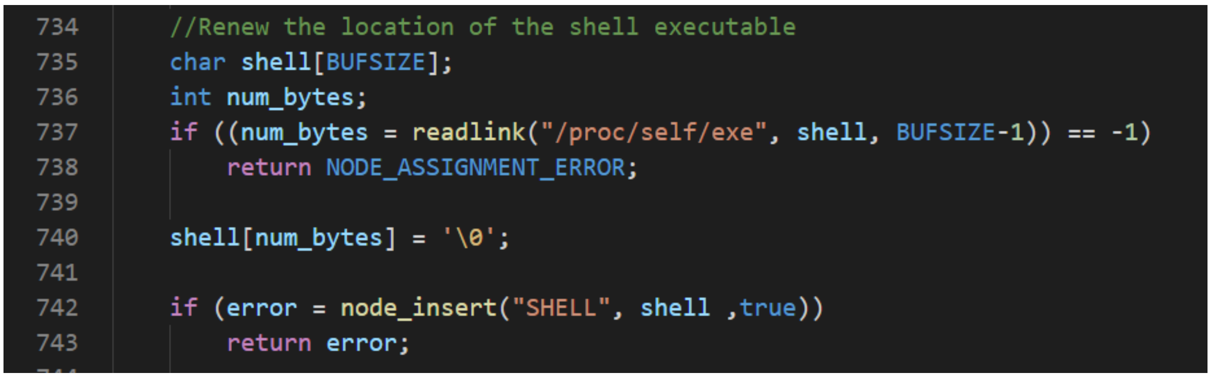


```
// Set every enviroment variable as a shell variable, with the bool env set to true.

for (char** env_var = environ; *env_var; env_var++)
{
    char var[TOKEN_SIZE];
    strncpy(var,*env_var,TOKEN_SIZE);

    char* key = strtok(var, "=");
    if (error = node_insert(key, getenv(key), true))
        return error;
}
```

If the following variables are not in the linked list, then they are initialised with the following default values.

- PATH = “bin/bash”
- HOME = the home directory, using *getpwuid()* function
- USER = the user name, using the *getpwuid()* function
- PROMPT = “init>”
- TERMINAL = the value of environment variable *TERM*, if set; Otherwise uses the return value of the function *ttyname* with 0 as an argument.
- EXITCODE = “NONE”

The shell variable *SHELL* is updated (or created) with its value pointing to the smash executable. This is done using the *readlink()* function. The value returned by this function is truncated to a maximum of BUFSIZE-1 bytes.

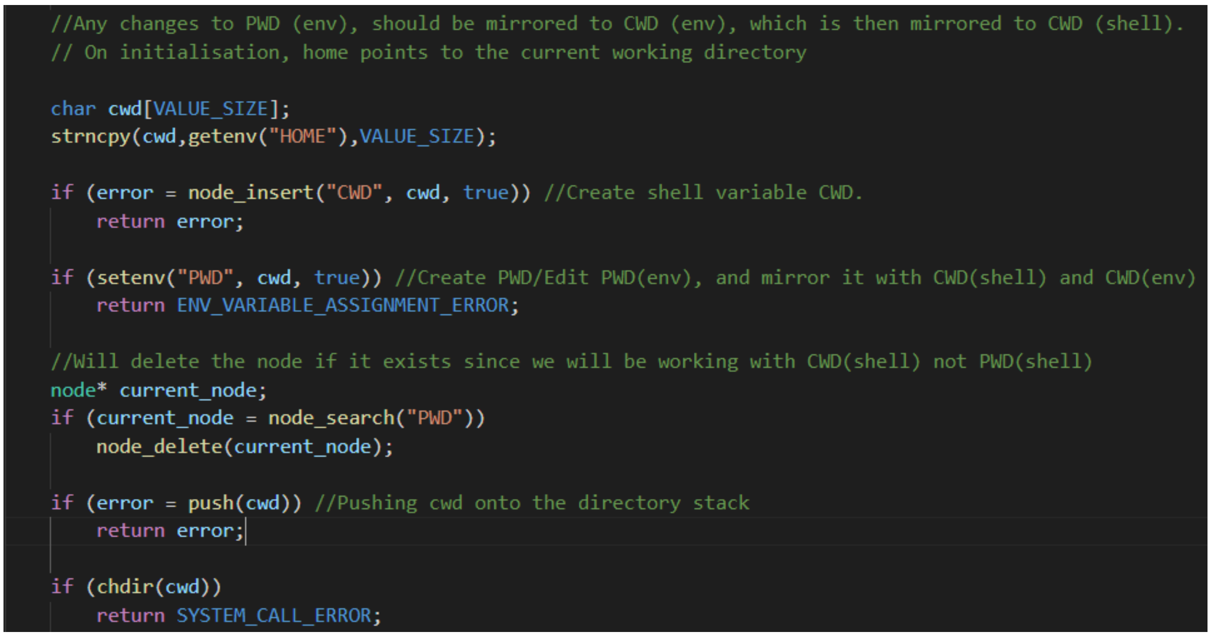


```
//Renew the location of the shell executable
char shell[BUFSIZE];
int num_bytes;
if ((num_bytes = readlink("/proc/self/exe", shell, BUFSIZE-1)) == -1)
    return NODE_ASSIGNMENT_ERROR;

shell[num_bytes] = '\0';

if (error = node_insert("SHELL", shell ,true))
    return error;
```

A new shell variable *CWD* is added to the linked list using the *node_insert* function with the *env* argument set to true. (This means that CWD will be added to the environment variables, if it doesn't already exist). The value of *CWD* is the value of the environment variable *HOME*. The value of the environment variable *HOME* is truncated to a maximum of *VALUE_SIZE* bytes.

Then the shell variable *PWD* is deleted from the linked list because *CWD* will be replacing it.

Finally, the value of the environment variable HOME is pushed to the directory stack and the current directory is changed to it using the *chdir()* function.



```
//Any changes to PWD (env), should be mirrored to CWD (env), which is then mirrored to CWD (shell).
// On initialisation, home points to the current working directory

char cwd[VALUE_SIZE];
strncpy(cwd,getenv("HOME"),VALUE_SIZE);

if (error = node_insert("CWD", cwd, true)) //Create shell variable CWD.
    return error;

if (setenv("PWD", cwd, true)) //Create PWD/Edit PWD(env), and mirror it with CWD(shell) and CWD(env)
    return ENV_VARIABLE_ASSIGNMENT_ERROR;

//Will delete the node if it exists since we will be working with CWD(shell) not PWD(shell)
node* current_node;
if (current_node = node_search("PWD"))
    node_delete(current_node);

if (error = push(cwd)) //Pushing cwd onto the directory stack
    return error;

if (chdir(cwd))
    return SYSTEM_CALL_ERROR;
```

The function returns 0 on success and an *err* instance on failure.

vars_valid

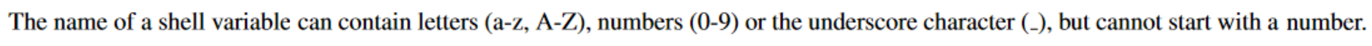
Function protocol: bool vars_valid(const char* token, int j)

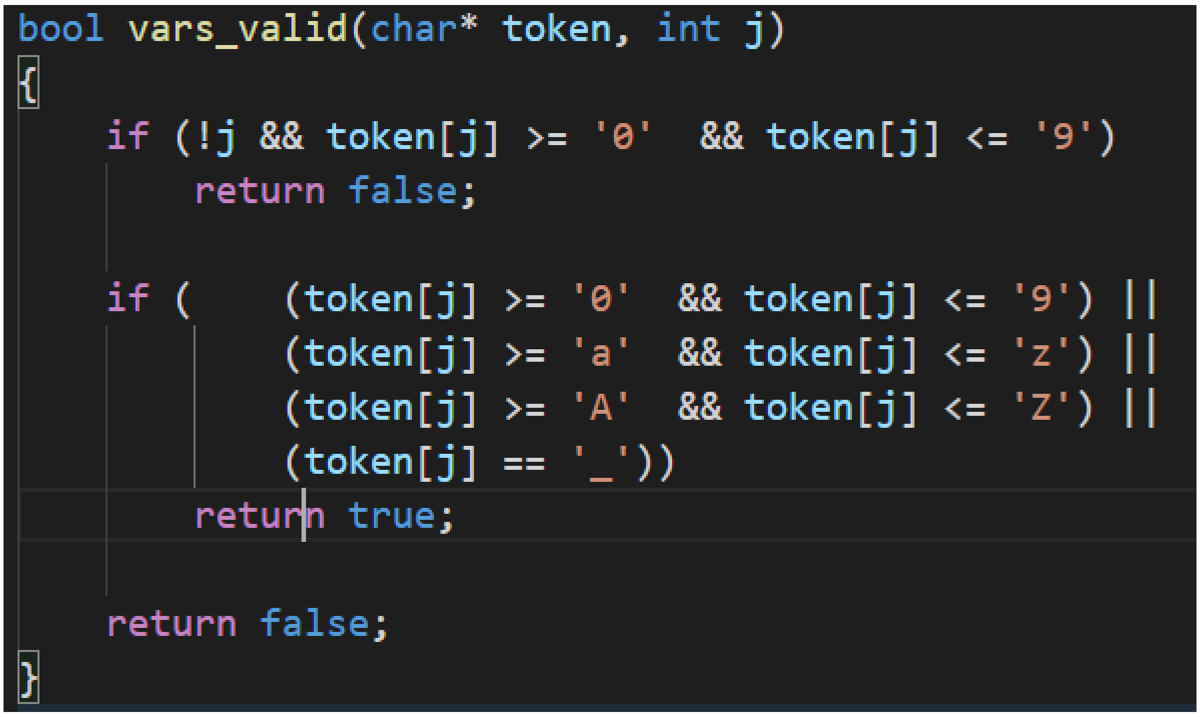
This function check if a given character at position *j* in *token* violates the conditions for a shell variable name. The conditions are shown below.

token is the string with the character to check.

j is the index of the character in *token* to check.

Condition:


The name of a shell variable can contain letters (a-z, A-Z), numbers (0-9) or the underscore character (_), but cannot start with a number.


```
bool vars_valid(char* token, int j)
{
    if (!j && token[j] >= '0'  && token[j] <= '9')
        return false;

    if (    (token[j] >= '0'  && token[j] <= '9') ||
            (token[j] >= 'a'  && token[j] <= 'z') ||
            (token[j] >= 'A'  && token[j] <= 'Z') ||
            (token[j] == '_'))
        return true;

    return false;
}
```

The function returns true if the character matches the condition, false otherwise.

expand_vars

Function protocol:

int expand_vars(char* tokens[TOKEN_SIZE], tokenchar_pair* var_indices, int var_indices_len, int m)

This function uses the indices in the mth element of *var_indices* to identify the Variable Expansion Characters within any string in *tokens*. Then the characters representing the variable name in a string are textually replaced with the value they're representing, if any.

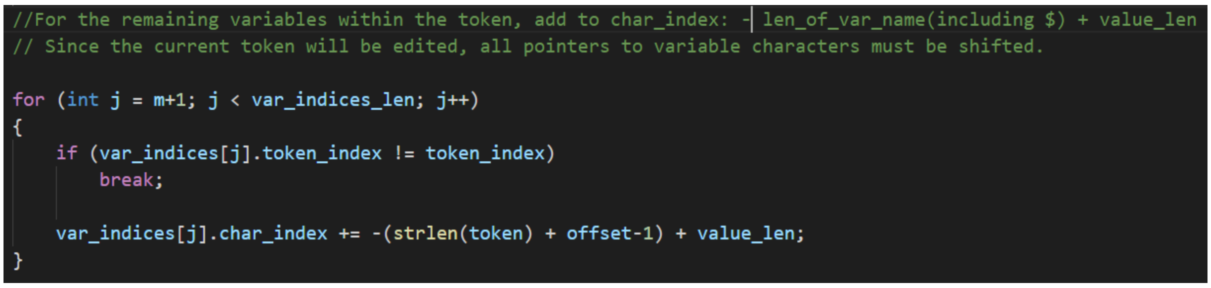
tokens is a double character pointer, where any variable names will be replaced by its textual value.

var_indices is an array of *tokenchar_pair* structs, representing the position of the Variable Expansion Character.

var_indices_len is the length of *var_indices*.

m is an index used to access *var_indices*.

Since this function replaces a variable name with its textual value, it means that any following characters in that string are going to be shifted. So at the end of this function all other remaining *tokenchar_pair* structs for that particular string are updated.



```
//For the remaining variables within the token, add to char_index: -len_of_var_name(including $) + value_len
// Since the current token will be edited, all pointers to variable characters must be shifted.

for (int j = m+1; j < var_indices_len; j++)
{
    if (var_indices[j].token_index != token_index)
        break;

    var_indices[j].char_index += -(strlen(token) + offset-1) + value_len;
}
```

This function determines the variable name even if it is not terminated by a meta character. See video demonstration for examples.

The function returns 0 on success and an *err* instance on failure.

assign_vars

Function protocol: err assign_vars(char** tokens, int length, int i, int k)

This function splits the token at position *i* on the character at position *k* and creates/updates a new shell variable with the LHS as a key and the RHS as the variable.

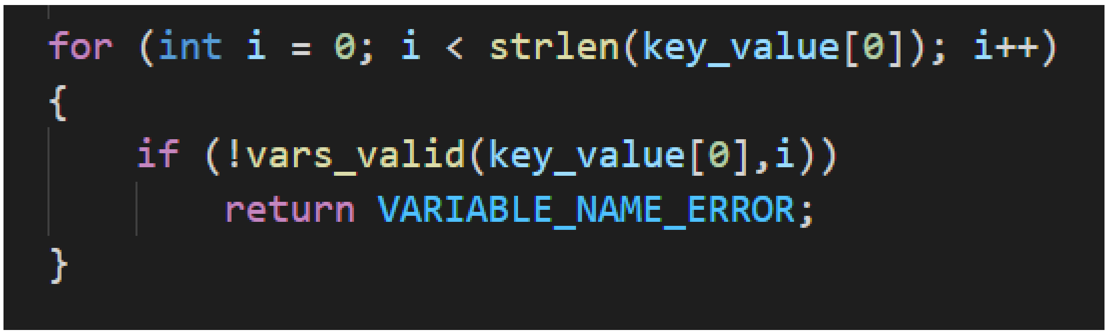
tokens is a double character pointer, where any variable assignment statements will be executed.

length is the number of strings in *tokens*.

i is the index of the String within *tokens* where variable assignment will take place.

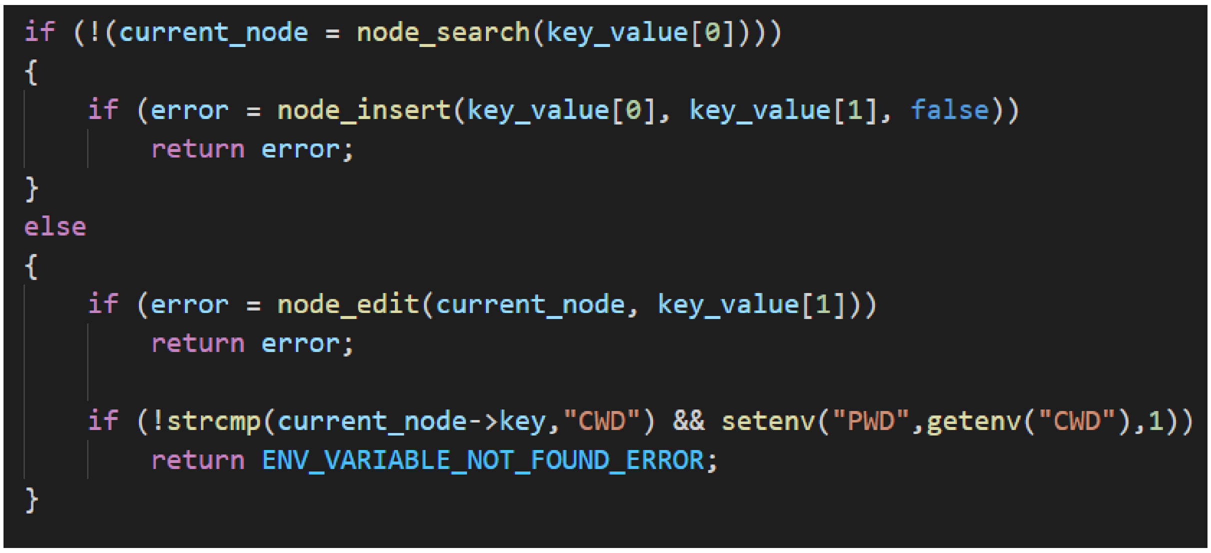
k is the index of the character within the token at position *I* where the first equals character (=) is identified.

The key must follow the conditions for a variable name and so each character is checked using the *vars_valid()* function.



```
for (int i = 0; i < strlen(key_value[0]); i++)
{
    if (!vars_valid(key_value[0],i))
        return VARIABLE_NAME_ERROR;
}
```

If the variable already exists, then the variable (and maybe its environment counter-part) is updated using the *node_edit()* function. If the variable edited was CWD, then the environment variable PWD is updated with the new value, since PWD and CWD are always mirrored.



```
if (!(current_node = node_search(key_value[0])))
{
    if (error = node_insert(key_value[0], key_value[1], false))
        return error;
}
else
{
    if (error = node_edit(current_node, key_value[1]))
        return error;

    if (!strcmp(current_node->key,"CWD") && setenv("PWD",getenv("CWD"),1))
        return ENV_VARIABLE_NOT_FOUND_ERROR;
}
```

The function returns 0 on success and an *err* instance on failure.

Commands

execute_internal

Function protocol: err execute_internal(char* args[TOKEN_SIZE], int arg_num, cmdno j)

This function considers the Strings in *args* as arguments to the *j*th internal command as defined in *internal_commands*.

args is an array of Strings which represent arguments

arg_num is the length of *args*

j is one of the definitions of *cmds* as defined in *headers.h*

This function can execute 11 different internal commands.

1. exit
2. echo
3. cd
4. showvar
5. export
6. unset
7. showenv
8. pushd
9. popd
10. dirs.
11. source

The program distinguishes between commands using a switch statement on *j*. The default case return a *BUFFER_OVERFLOW_ERROR*, because the program can be interpreted as trying to access an element in *internal_commands* that doesn't exsist.

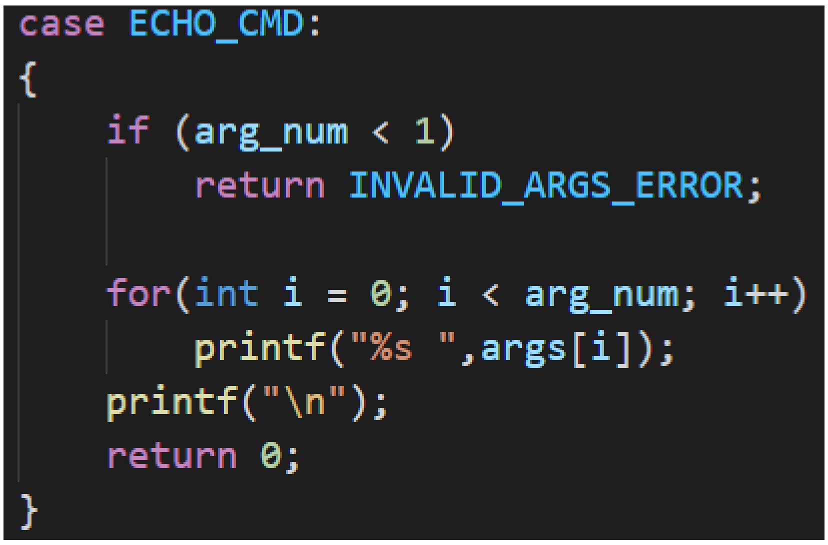
Exit

This *exit* command sets *exit_value* and sets *exit_program* to true. *exit* only accepts one argument or none at all. The argument provided is used to set *exit_value*. To exit the program the programmer must continually check this value as done in *smash.c line 42*.

The function returns 0 on success and an *err* instance on failure.

Echo

The *echo* command prints all arguments provided to standard output. The arguments must be greater than 0. It simply loops over all the arguments and outputs them separated by a single white space character. Then a new line character is added to the end.

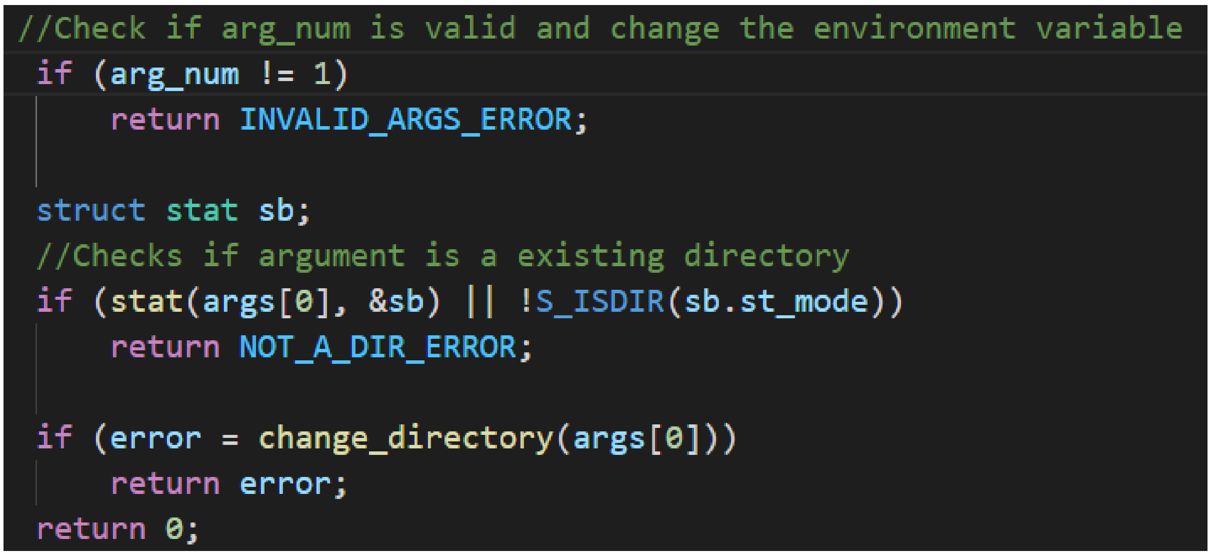


```
case ECHO_CMD:
{
    if (arg_num < 1)
        return INVALID_ARGS_ERROR;

    for(int i = 0; i < arg_num; i++)
        printf("%s ",args[i]);
    printf("\n");
    return 0;
}
```

Cd

cd changes the current working directory if the directory provided is acceptable and updates the directory stack as necessary. Only one argument is accepted. The program checks if the directory exists using the *stat* struct with the functions *stat()* and S_ISDIR() as shown below. If valid, the function then calls *change_directory().*

The function returns 0 on success and an *err* instance on failure.

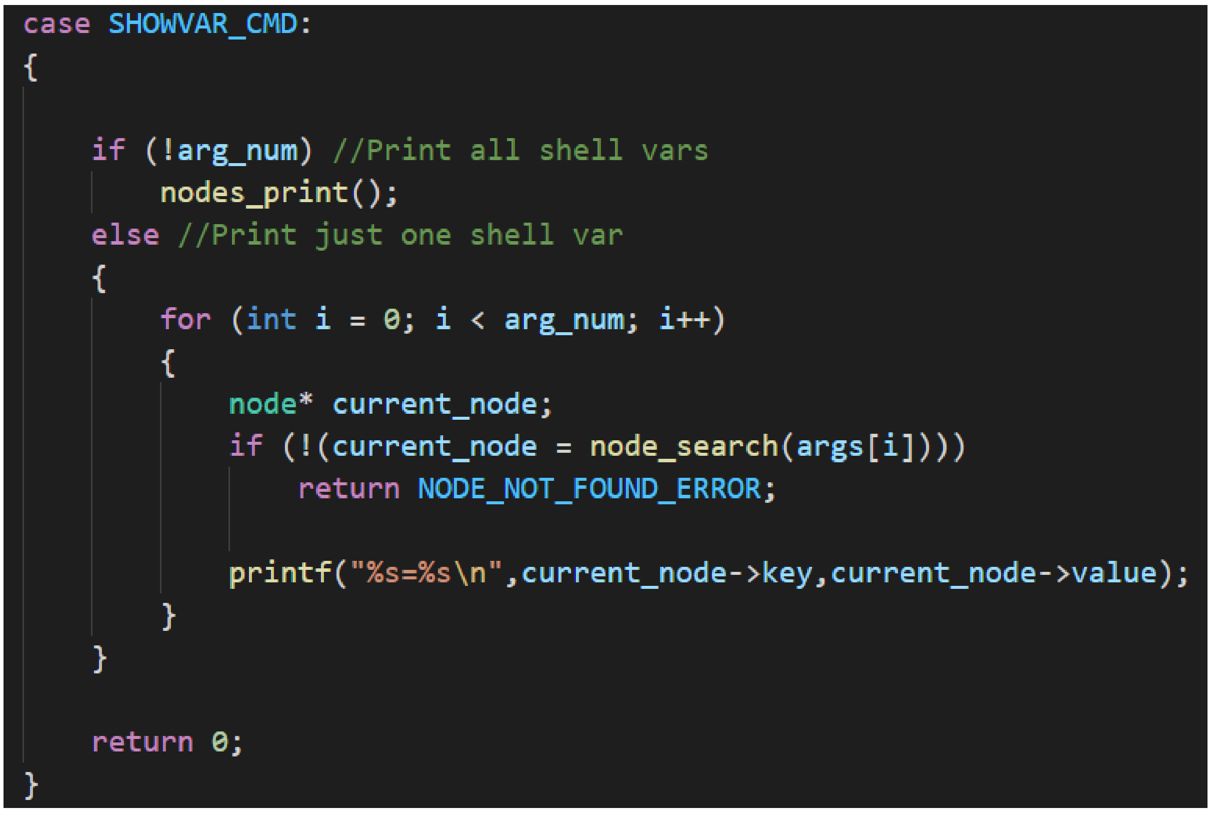


```
//Check if arg_num is valid and change the environment variable
 if (arg_num != 1)
     return INVALID_ARGS_ERROR;

 struct stat sb;
 //Checks if argument is a existing directory
 if (stat(args[0], &sb) || !S_ISDIR(sb.st_mode))
     return NOT_A_DIR_ERROR;

 if (error = change_directory(args[0]))
     return error;
 return 0;
```

Showvar

showvar prints variable key value pairs in the shell variable doubly linked list. If argument are provided then the keys are searched for in the linked list and if found the corresponding key value pair is printed to standard output. If a key is not found to be in the linked list the function exits and any other variables are not printed. If no argument is provided then all key value pairs are printed using *nodes_print()*.

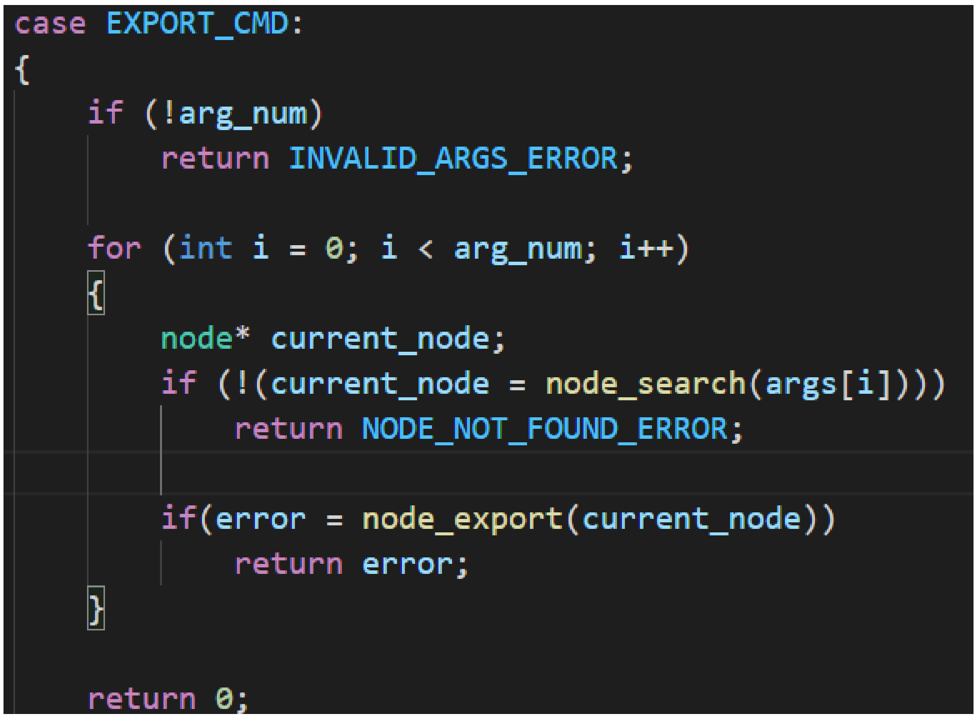
The function returns 0 on success and an *err* instance on failure.



```
case SHOWVAR_CMD:
{

    if (!arg_num) //Print all shell vars
        nodes_print();
    else //Print just one shell var
    {
        for (int i = 0; i < arg_num; i++)
        {
            node* current_node;
            if (!(current_node = node_search(args[i])))
                return NODE_NOT_FOUND_ERROR;

            printf("%s=%s\n",current_node->key,current_node->value);
        }
    }

    return 0;
}
```

Export

export finds the keys corresponding to its arguments and sets that item in the shell variable doubly linked list's *env* variable to true. The argument count must be greater than 0. If a key which is not in the linked list is found then the program returns a positive integer and any other following keys are not considered.

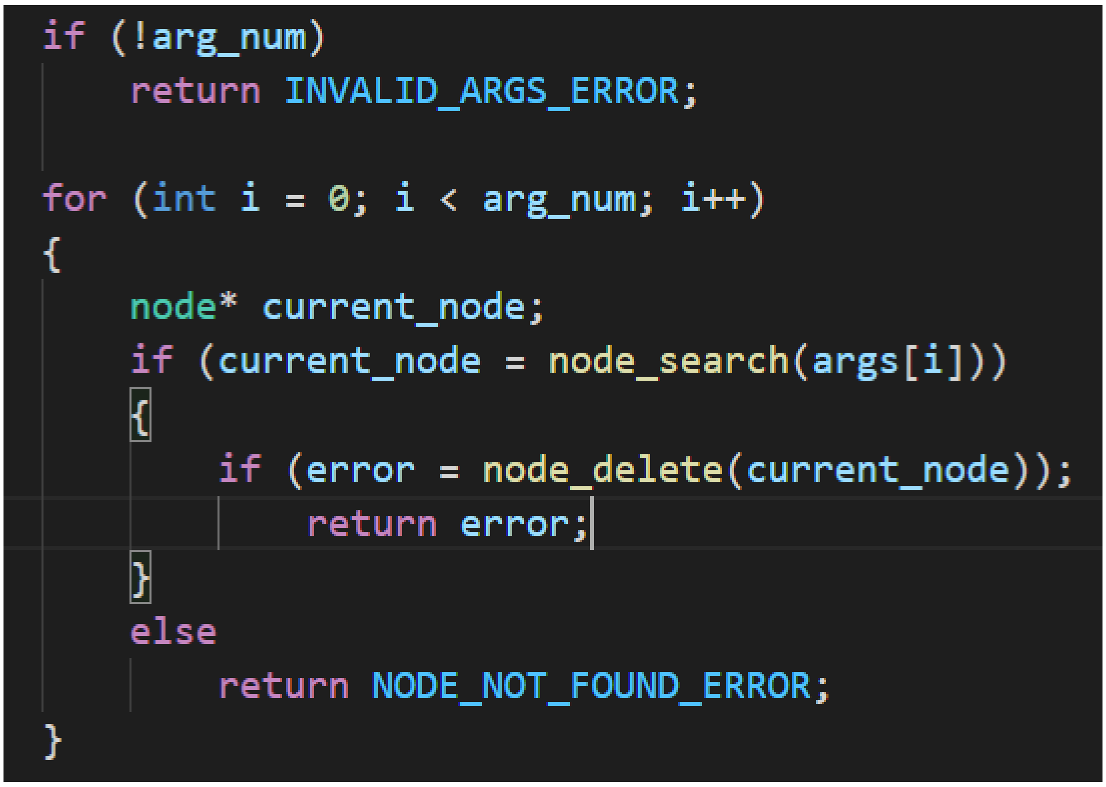


```
case EXPORT_CMD:
{
    if (!arg_num)
        return INVALID_ARGS_ERROR;

    for (int i = 0; i < arg_num; i++)
    {
        node* current_node;
        if (!(current_node = node_search(args[i])))
            return NODE_NOT_FOUND_ERROR;

        if(error = node_export(current_node))
            return error;
    }

    return 0;
```

The function returns 0 on success and an *err* instance on failure.

Unset

unset removes the variables corresponding to its arguments from the shell variable linked list. The argument count must be greater than 0. A for loops iterates over every argument and calls *node_delete* if that variable exists. Otherwise the function exits on the variable that doesn't exist and any other following arguments are not considered.

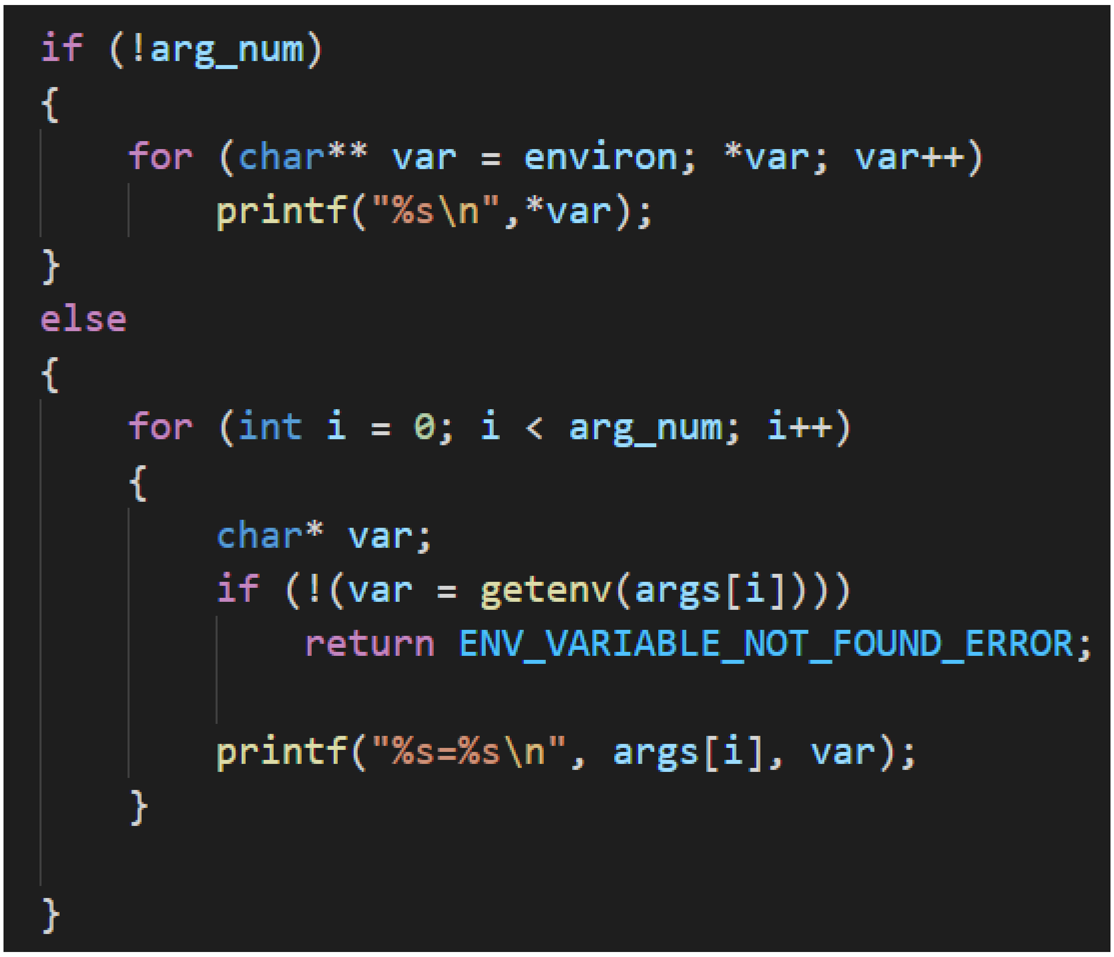


```
if (!arg_num)
    return INVALID_ARGS_ERROR;

for (int i = 0; i < arg_num; i++)
{
    node* current_node;
    if (current_node = node_search(args[i]))
    {
        if (error = node_delete(current_node));
            return error;
    }
    else
        return NODE_NOT_FOUND_ERROR;
}
```

The function returns 0 on success and an *err* instance on failure.

Showenv

showenv prints environment variables to standard output. If no arguments are specified, all the environment variables are printed in key-value pairs. Otherwise, the program uses its arguments to search for a matching key using *getenv()*, and output that key-value pair. If a key that is not in the shell variable linked list is encountered, the function exits and any other arguments are not considered.

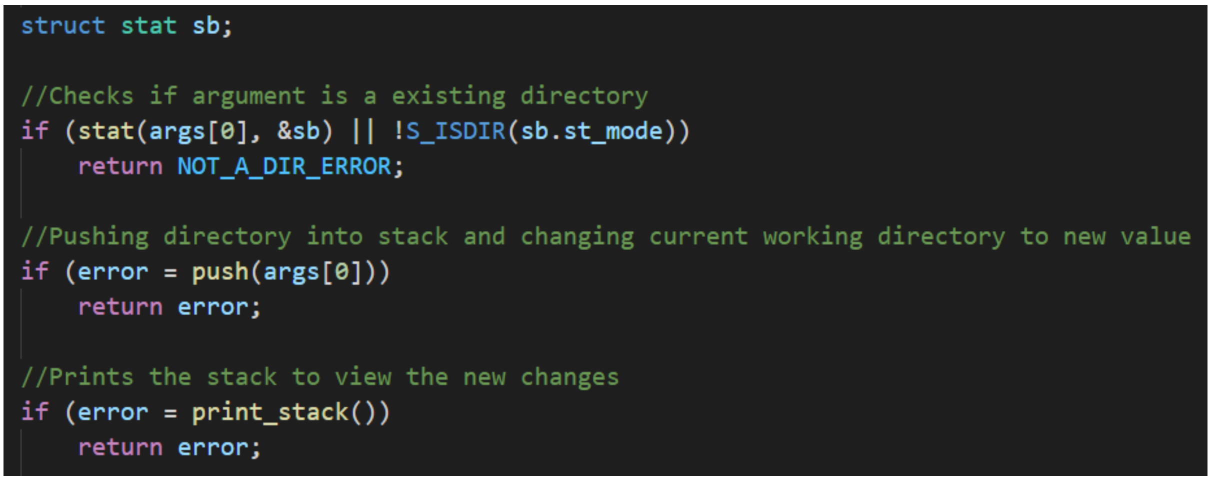


```c
if (!arg_num)
{
    for (char** var = environ; *var; var++)
        printf("%s\n",*var);
}
else
{
    for (int i = 0; i < arg_num; i++)
    {
        char* var;
        if (!(var = getenv(args[i])))
            return ENV_VARIABLE_NOT_FOUND_ERROR;

        printf("%s=%s\n", args[i], var);
    }

}
```

The function returns 0 on success and an *err* instance on failure.

Pushd

pushd pushes a valid directory into the directory stack. It only takes on argument, the directory to push. The program checks if the directory exists using the *stat* struct with the functions *stat()* and S_ISDIR() as shown below. Then that directory is pushed into the stack using *push()* and the directory stack is printed to standard output using *print_stack()*.

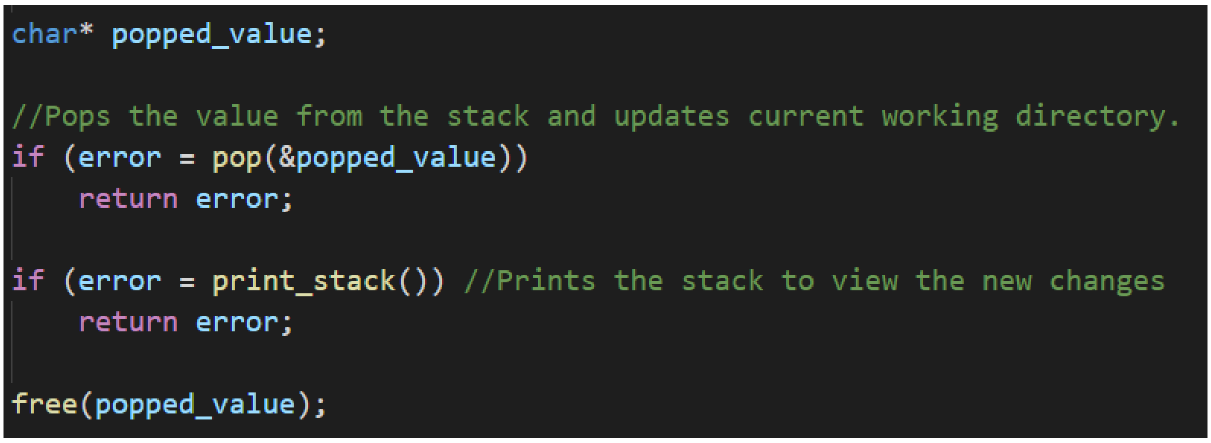


```c
struct stat sb;

//Checks if argument is a existing directory
if (stat(args[0], &sb) || !S_ISDIR(sb.st_mode))
    return NOT_A_DIR_ERROR;

//Pushing directory into stack and changing current working directory to new value
if (error = push(args[0]))
    return error;

//Prints the stack to view the new changes
if (error = print_stack())
    return error;
```

The function returns 0 on success and an *err* instance on failure.

Popd

popd pops the top most item from the stack provided that it is not the only item present in the stack. This function takes no arguments. The value is popped using *pop()* and then the stack is printed to standard output using *print_stack()*.

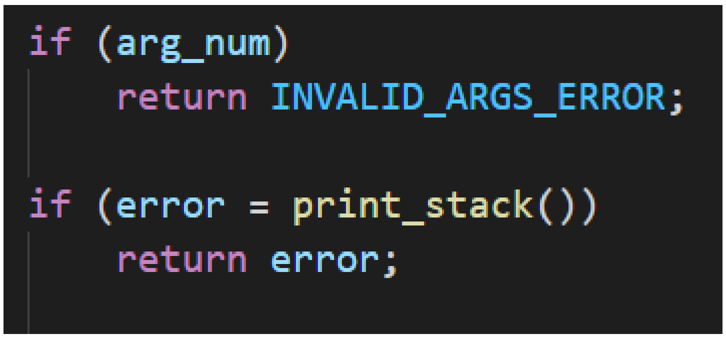


```c
char* popped_value;

//Pops the value from the stack and updates current working directory.
if (error = pop(&popped_value))
    return error;

if (error = print_stack()) //Prints the stack to view the new changes
    return error;

free(popped_value);
```

The function returns 0 on success and an *err* instance on failure.

Dirs

dirs prints the directory stack and takes no arguments. This is done by calling *print_stack()*.

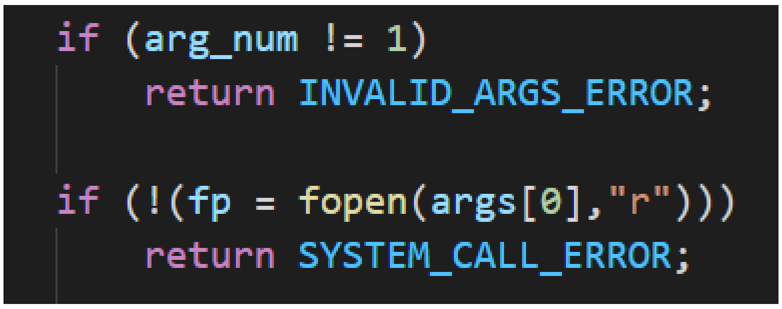


```c
if (arg_num)
    return INVALID_ARGS_ERROR;

if (error = print_stack())
    return error;
```

The function returns 0 on success and an *err* instance on failure.

Source

source identifies a file to get input from, one line every iteration of the program. It only takes one argument. The file name is opened using *fopen()* on read only mode and the return value is assigned to *fp*.



```
if (arg_num != 1)
    return INVALID_ARGS_ERROR;

if (!(fp = fopen(args[0],"r")))
    return SYSTEM_CALL_ERROR;
```

The function returns 0 on success and an *err* instance on failure.

execute_external

Function protocol: err execute_external(char** tokens, redirect_ext* ex)

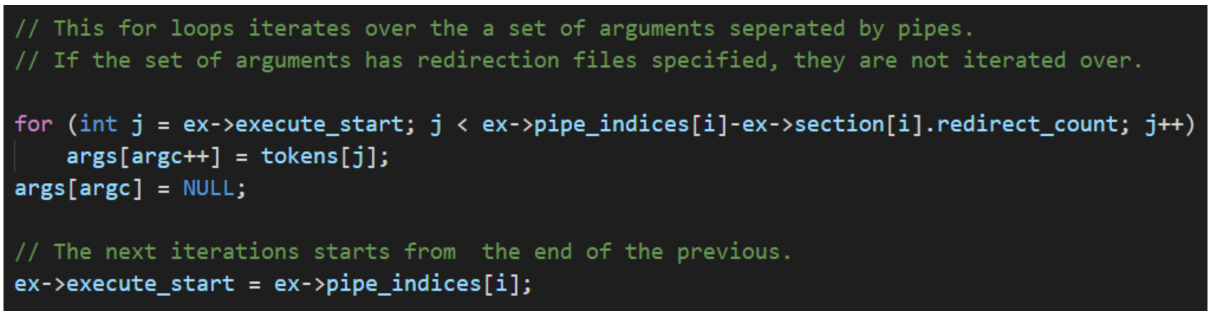
This function executes external commands via the *fork* + *exec* pattern. It facilitates piping between commands, and commands can be redirected to/from certain files between pipes, albeit the fact this would stop the pipe chain. (See video demonstration for examples).

tokens is the array of Strings holding the individual tokens

ex contains information about the token space, and this should be filled from *tokens_init*.

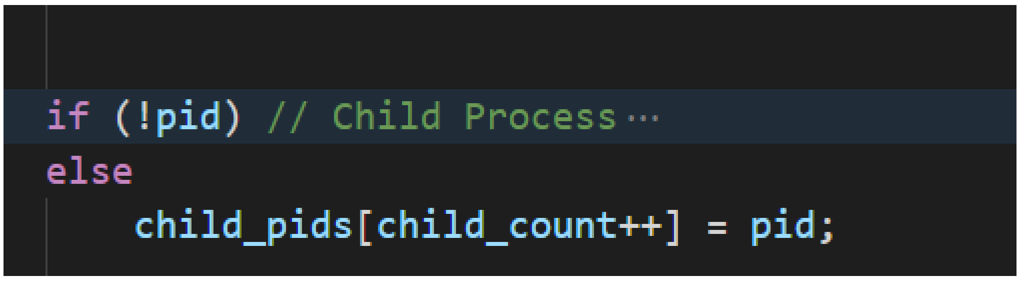
The program iterates over every chunk of tokens separated by pipes.

Each individual chunk may contain at most one command, arguments, and input and output redirects. The tokens in the current chunk are stored in an array and the number of tokens is tallied, for further use. Note that the tokens used to specify redirects are not considered.



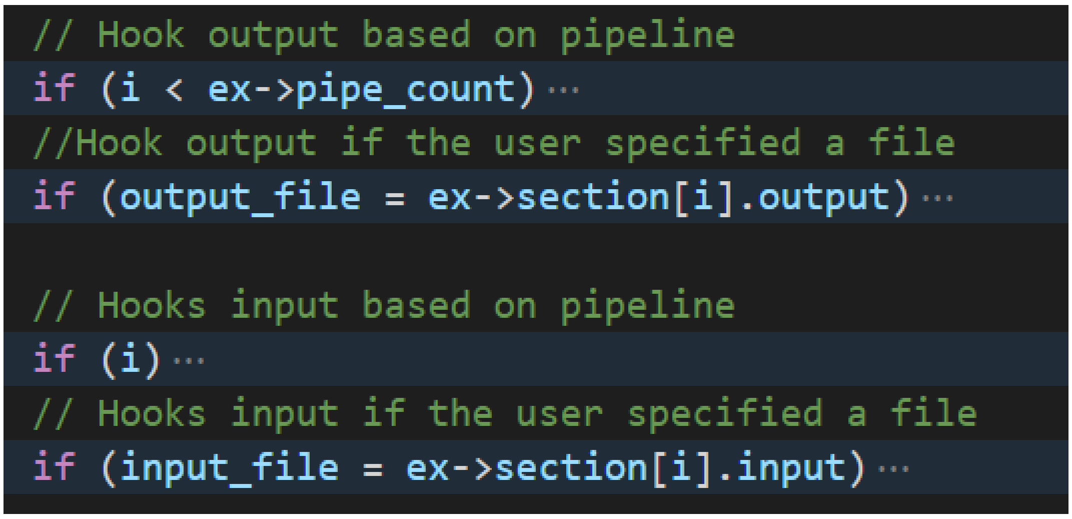
```
// This for loops iterates over the a set of arguments seperated by pipes.
// If the set of arguments has redirection files specified, they are not iterated over.

for (int j = ex->execute_start; j < ex->pipe_indices[i]-ex->section[i].redirect_count; j++)
    args[argc++] = tokens[j];
args[argc] = NULL;

// The next iterations starts from  the end of the previous.
ex->execute_start = ex->pipe_indices[i];
```

On each iteration, the program forks a child, and that is tallied in *child_pids*. Note that *child_count* can never exceed *BUFSIZE* which is the length of *child_pids*. This is because the number of children is dependent on the number of *pipe* characters, and in *tokens_init* a maximum of *BUFSIZE* is allowed.



```
if (!pid) // Child Process···
else
    child_pids[child_count++] = pid;
```

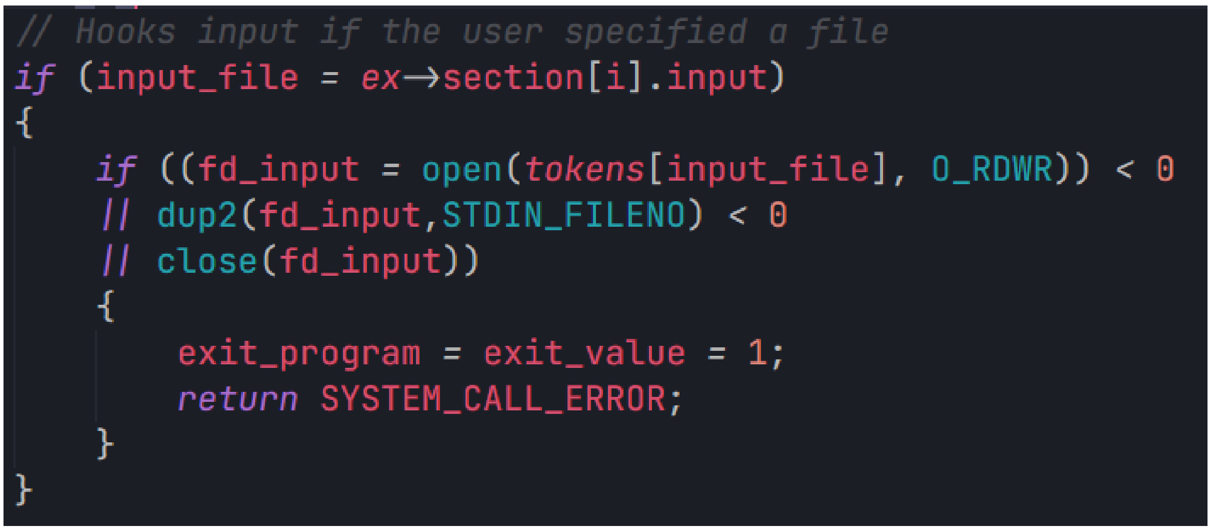
The child process then hooks its input and output streams based on its position in the pipe. The user may choose to manually redirect to a file, although this will break the pipe chain. The element at position *i* in *section's* values are checked and if any redirect is specified (ie. if *input* or *output* is set), then the appropriate stream is redirected to the filename pointed to by either *input* or *output*.



```
// Hook output based on pipeline
if (i < ex->pipe_count)···
//Hook output if the user specified a file
if (output_file = ex->section[i].output)···

// Hooks input based on pipeline
if (i)···
// Hooks input if the user specified a file
if (input_file = ex->section[i].input)···
```

A stream is hooked based on pipelines by calling *dup2* with the appropriate file descriptor on standard input/output.

A stream is hooked based on user specified files by first opening that file using open, then using *dup2* with the just opened file descriptor on standard input/output.

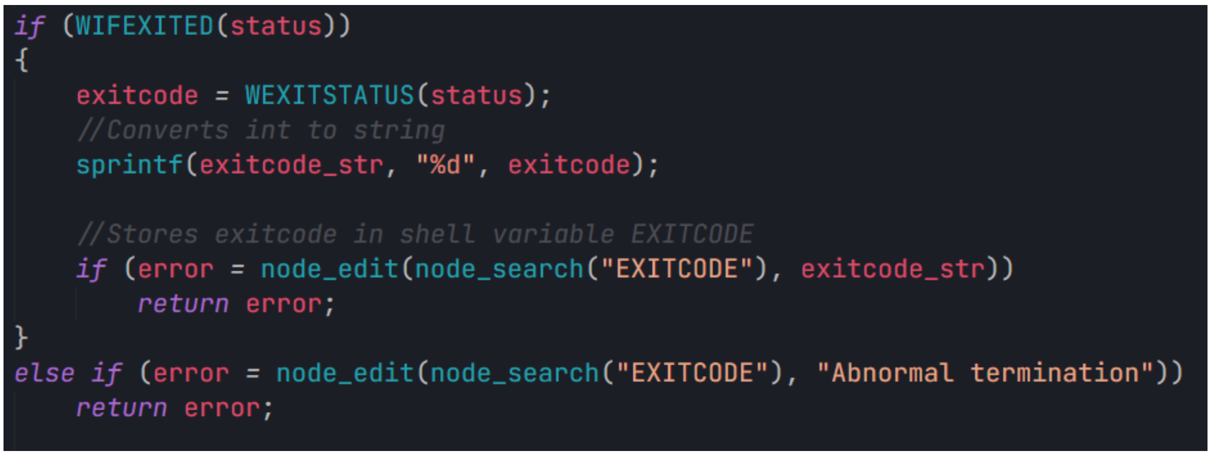


```
// Hooks input if the user specified a file
if (input_file = ex->section[i].input)
{
    if ((fd_input = open(tokens[input_file], O_RDWR)) < 0
    || dup2(fd_input,STDIN_FILENO) < 0
    || close(fd_input))
    {
        exit_program = exit_value = 1;
        return SYSTEM_CALL_ERROR;
    }
}
```

The token at *input_file* is opened with the flags *O_RDWR*.

Finally the child processes' binary image is replaced by the call to *execvp()* with the current token chunk passed in. If this fails *perror()* shows an appropriate error message and the program exits with exit code 1.

After this for loop finishes iterating over every token chunk, the program waits for all child processes to finish. Afterwards, it takes the return code of the last program to exit and sets it to the shell variable *EXITCODE*, if it exists. Otherwise "*Abnormal Termination*" is saved.



```
if (WIFEXITED(status))
{
    exitcode = WEXITSTATUS(status);
    //Converts int to string
    sprintf(exitcode_str, "%d", exitcode);

    //Stores exitcode in shell variable EXITCODE
    if (error = node_edit(node_search("EXITCODE"), exitcode_str))
        return error;
}
else if (error = node_edit(node_search("EXITCODE"), "Abnormal termination"))
    return error;
```

The function returns 0 on success and an *err* instance on failure.

Redirects for internal commands

handle_redirect

Function Protocol: err handle_redirect(char** tokens, int state, int j, redirect_int* in)

This function fills the struct *in's input_filename, output_filename* and *ouput_cat_filename* based on the given value of *state.*

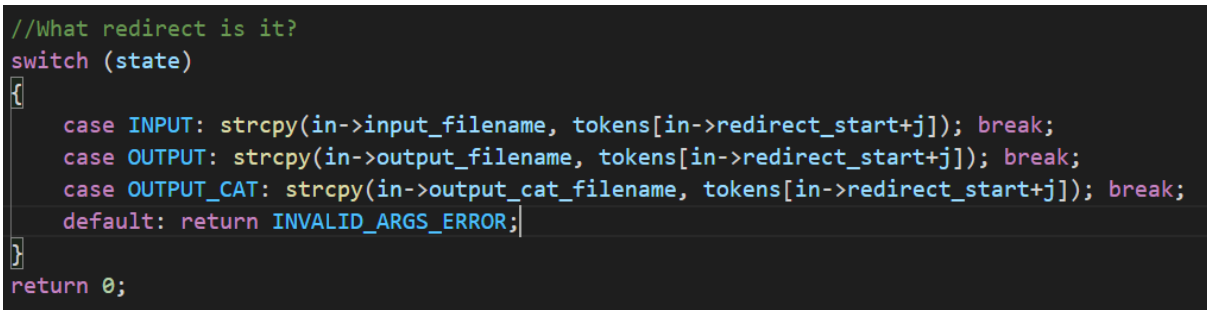
tokens an array of Strings.

state refers to the type of redirect character the function is dealing with

j refers to an offset used to access the filename token in *tokens*.

in is the struct where the *input_filename, output_filename* and *ouput_cat_filename* will be filled.

The function accesses the token at *redirect_start* and offsets it by j. *redirect_start* points to the token with the first filename, and j is used as an offset to get any other subsequent filename tokens. The filenames are then copied to their respective variable.



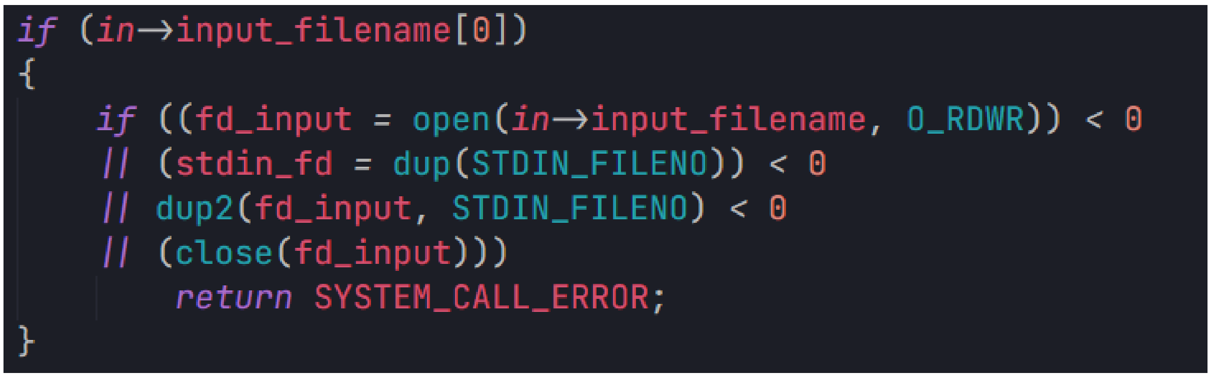
```
//What redirect is it?
switch (state)
{
    case INPUT: strcpy(in->input_filename, tokens[in->redirect_start+j]); break;
    case OUTPUT: strcpy(in->output_filename, tokens[in->redirect_start+j]); break;
    case OUTPUT_CAT: strcpy(in->output_cat_filename, tokens[in->redirect_start+j]); break;
    default: return INVALID_ARGS_ERROR;
}
return 0;
```

The function returns 0 on success and an *err* instance on failure.

hook_streams

Function protocol: err hook_streams(const redirect_int* in)

This function uses the struct *in* to open files and hook standard input/output to the filenames specified. Before doing so, *stdin_fd/stdout_fd* is set to the file descriptor of the current standard input/output. This is so they can be reverted back in *reset_streams()*.



```
if (in->input_filename[0])
{
    if ((fd_input = open(in->input_filename, O_RDWR)) < 0
    || (stdin_fd = dup(STDIN_FILENO)) < 0
    || dup2(fd_input, STDIN_FILENO) < 0
    || (close(fd_input)))
        return SYSTEM_CALL_ERROR;
}
```

input_filename is opened with the flags *O_RDWR*

ouput_filename is opened with the flags *O_CREAT | O_RDWR | O_TRUNC* and has user permissions *S_IRWXU*

output_cat_filename is opened with the flags *O_CREAT | O_APPEND | O_RDWR* and has user permissions *S_IRWXU*

This function gives precedence to the output type that came last. (eg. echo ABC > a >> a) → *'ABC'* will be appended to *a*, not truncated.

Miscellaneous

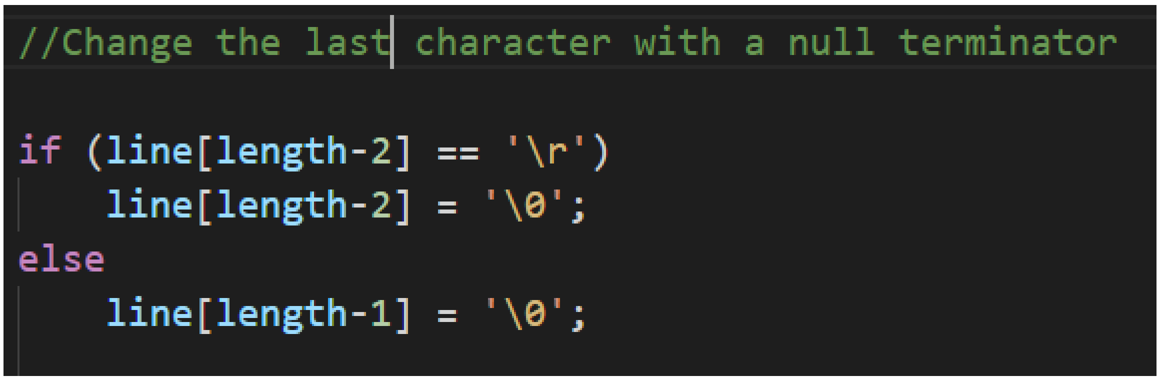
get_input_from_file

Function protocol: char* get_input_from_file(FILE* fp)

This function reads one line from the file pointed to by *fp* (*not the global variable, but the parameter*), and returns said line.

fp is a file pointer to a file to be read from.

The function dynamically allocates memory to the return value and hence should be freed afterwards. It reads a maximum of *BUFSIZ* (~8KB) characters per line. All lines are terminated by a NULL terminator and in the case the carriage return character is read at the end, the carriage return is substituted by a NULL terminator.



```
//Change the last character with a null terminator

if (line[length-2] == '\r')
    line[length-2] = '\0';
else
    line[length-1] = '\0';
```

The function returns a pointer to a dynamically allocated string on success and NULL if EOF in *fp* is reached, or an error is encountered in which case *error* is appropriately set.

contains_char

Function protocol: int contains_char(const char* string, char a)

This function checks wether the character *a* is contained in *string* and returns the position of the first instance of that character.

string is a String to search through

a is the character to search for in *string*.

If the character does not exist, then -1 is returned, otherwise the position of the first instance of *a* within *string* is returned.

str_to_int

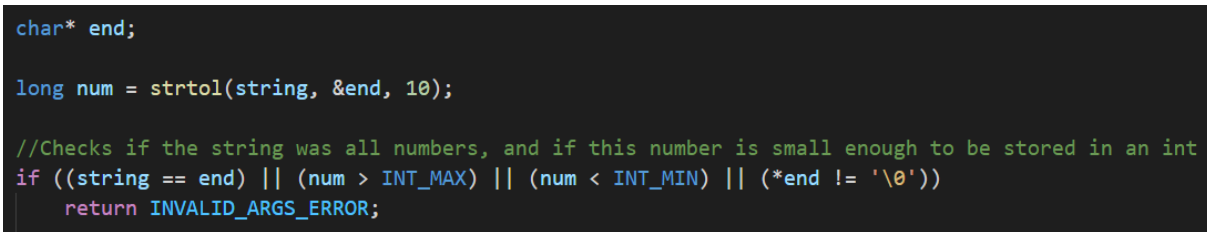
Function protocol: err str_to_int(int* value, const char* string)

This function transforms *string* into its integer equivalent.

value a memory location where the integer value of *string* is to be saved

string is a String to get the integer value from.

At most, 10 characters are read from *string*. *string* must be made up of all numbers and no other characters. Moreover the number extracted must fit in an *int* data type.



```c
char* end;

long num = strtol(string, &end, 10);

//Checks if the string was all numbers, and if this number is small enough to be stored in an int
if ((string == end) || (num > INT_MAX) || (num < INT_MIN) || (*end != '\0'))
    return INVALID_ARGS_ERROR;
```

The function returns 0 on success and an *err* instance on failure.

contains_word

Function protocol: int contains_word(const char* input, const char* key)

This function checks whether *key* is subsumed by *input*.

input is a string to search through.

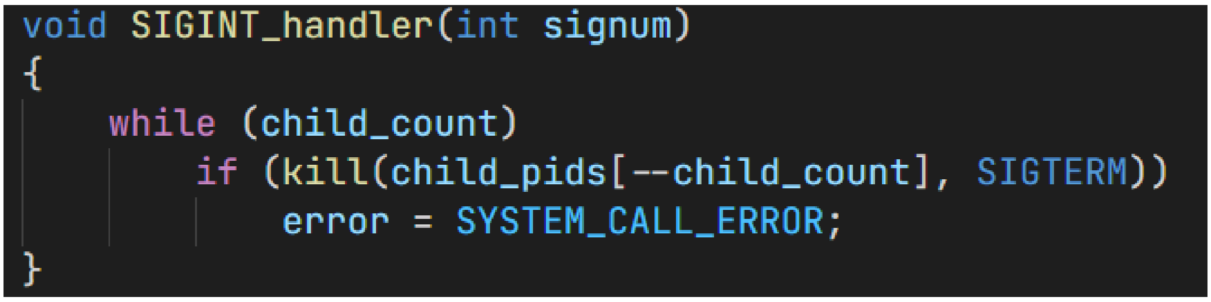
key is a string to find in *input*.

The function returns *true* if *key* is subsumed by *input*, *false* otherwise.

SIGINT_handler

Function protocol: void SIGINT_handler(int signum)

This function is a signal handler for SIGINT. It sends SIGTERM to all child processes via the *kill()* function. It iterates over *child_pids* and sets *child_count* to 0. All variables in this function have the *volatile* key word.



```c
void SIGINT_handler(int signum)
{
    while (child_count)
        if (kill(child_pids[--child_count], SIGTERM))
            error = SYSTEM_CALL_ERROR;
}
```