

# Operating Systems & Systems Programming

---

CPS 1012

Tristan Oa Galea

# **Table of Contents**

Introduction.....	pg 2
Shell Variables.....	pg 3
Command-line Interface.....	pg 8
Internal Commands.....	pg 9
External Commands.....	pg 16
Redirection.....	pg 19
Piping.....	pg 30
Process Management.....	pg 36
Testing.....	pg 38

# **Introduction**

In this assignment, a simple command-line interpreter for Linux entitled 'EGGSHELL' was created.

This eggshell was programmed to be able to perform a number of operations including:

- Managing Shell Variables
- Computing of both internal and external commands
- Implementation of input and output redirection
- Implementation of single and multiple pipes
- Handling of interrupt signals (Process Management)

All these operations were performed successfully with the required error checking for each individual step. All bugs detected in testing of the shell were eliminated to improve the overall performance of the shell.

To sum this all up, the created shell was built up step by step and shaped to create a Linux shell which fits all the specified requirements in the assignment specification.

# Design of the System

## Shell Variables

To start off, shell variables were implemented first. Research was conducted on environmental variables and it was clear that certain variables required to be included in the system were already inherited from the previous process. These variables included the PATH, USER, HOME and SHELL environmental variables. Thus, nothing was done to these variables as they already existed.

It was then clear that ways of creating the PROMPT, CWD, TERMINAL and EXITCODE shell variables had to be identified.

To do this, 2 functions were created called `init_shell_vars()` and `set_exitcode()`. These functions were required to be called from the main function when the eggshell starts up.

In the **`init_shell_vars()`** function, the PROMPT, CWD and TERMINAL shell variables were created. These were done as follows:

The PROMPT shell variable was initialized using the inbuilt `setenv()` function initially setting the prompt to '>>>'. Upon unsuccessful assignment (if `setenv()` returned -1), the user is displayed with a prompt and -1 is returned. This method for error checking was also implemented for the other 3 remaining shell variables to be created.

```
/* Creates shell variable PROMPT and handles errors if unsuccessful */
if (setenv("PROMPT", ">>> ", 1) == -1) {
    perror("Error when setting PROMPT");
    return -1;
}
```

The CWD shell variable was created by first obtaining and storing the contents of the already existing PWD environmental variable to a character pointer. This directory obtained is then attempted to be opened and if successful, `setenv()` is used once again used to set CWD to that existing directory name. Validation is performed once again as explained before. If opening the directory fails, the user is informed and -1 is returned back.

```
/* Gets environmental variable PWD and stores in char pointer */
char *dirname = getenv("PWD");

/* Opens directory obtained */
DIR *dir = opendir(dirname);

/* If directory exists this section runs */
if (dir) {

    /* Opened directory is closed */
    closedir(dir);

    /* The CWD shell var is set with error checking included */
    if (setenv("CWD", dirname, 1) == -1) {
        perror("Error when setting CWD ");
        return -1;
    }

    /* If directory does not exist, user is informed
     * through an error message and function returns -1*/
} else {
    perror("Error when locating specified dir ");
    return -1;
}
```

The last variable to be created in this function is the `TERMINAL` shell variable. This was created by first ensuring that the content of the first argv is actually a terminal. If not the user is informed and -1 is returned. Moving on, the terminal name of the existing terminal is retrieved to be assigned to `TERMINAL` when using `setenv()`. If it is not successfully set, the user is informed like before and -1 is returned.

```
/* Creates shell variable TERMINAL and handles errors if unsuccessful */  
  
/* Converts the first argv element to an integer and stores in fd */  
int fd = atoi(argv[0]);  
  
/* If the value of fd is actually a terminal, this section runs */  
if (isatty(fd)) {  
  
    /* terminal name of existing terminal is stored */  
    char *tname = ttyname(fd);  
  
    /* TERMINAL shell var is set and error checking is performed */  
    if (setenv("TERMINAL", tname, 1) == -1) {  
        perror("Error when setting TERMINAL");  
        return -1;  
    }  
  
    /* If fd does not store existing terminal info, user is informed  
    * with an error and program execution returns to main with -1*/  
} else {  
    perror("Error when checking if terminal exists");  
    return -1;  
}
```

If all 3 variables are created successfully, 0 is returned back. It was taken as convention throughout the whole assignment that 0 signifies success and -1 or EXIT\_FAILURE signify a failed attempt. Thus for the EXITCODE to be set, a separate function for it had to be created through which the return value of the previous operation performed was passed as the only parameter.

```
/* A call to the functions which create the remaining 4 shell  
 * variables unique to the eggshell as required  
 * The return value of the init_shell_vars() function  
 * determines the value the EXITCODE shell var will have*/  
set_exitcode(init_shell_vars(argv));
```

In the **set\_exitcode()** function, the shell variable EXITCODE was set. This was done by setting the newly created EXITCODE to 0 with setenv() if the return value is 0. If not, the if statement checks for a return value of -125 and if so, enters that branch. The reason why this branch was included will be explained further on when discussing the all internal command. The else branch simply sets using setenv() the value of EXITCODE to the errno value set. Error checking as explained before was also performed.

```

/* If return value of previous function is 0, this code runs */
if (function == 0) {

    /* EXITCODE set to 0 and error checking is performed */
    if(setenv("EXITCODE", "0", 1) == -1){
        perror("setenv() fail");
        EXIT_FAILURE;
    }

    /* if return value is -125, run this part */
} else if (function == -125) {

    //do nothing

    /* if return value is not 0 nor -125, run this part */
} else {

    /* EXITCODE set to errno value and error checking is performed */
    if(setenv("EXITCODE", temp, 1) == -1){
        perror("setenv() fail");
        EXIT_FAILURE;
    }
}
}

```

Now that all the shell variables related to the eggshell have been created and set, a function which allows the user to set new shell variables or modify existing ones was created.

This was done by searching for an '=' in the first argument entered. If this is detected, code was written to split up the first part of the argument up till = to a char string name and the second part to a char string value.

```

719      /* If = is detected in the first argument */
720      if (strstr(args[0], "=") != NULL) {
721
722          /* 2 char pointers are initialised */
723          char *name;
724          char *value;
725
726          /* name stores the first part of the arg up till the = */
727          name = strtok(args[0], "=");
728
729          /* value stores the second part of the arg after the = */
730          /* Error checking is performed to ensure the user entered a value */
731          if ((value = strtok(NULL, "=")) == NULL) {
732              perror("No value entered");
733              return -1;
734          }
735      }

```

Before proceeding to storing the new shell variable based on the name and value entered, more code was added to check if a \$ was present in the value

string and if so, textual replacement of this code was done. This was done by using `getenv()` to search for a variable with that particular name and if this exists, it replaces the text entered in the value variable with the actual value stored in the entered variable.

```
736      /* If value entered has a dollar sign as the first character */
737      if (value[0] == '$') {
738
739          /* this is replaced with the corresponding value it is storing */
740          if ((value = getenv(&value[1])) == NULL) {
741              perror("getenv() fail ");
742              return -1;
743          }
744      }
745  }
```

Now that all these cases were outlined, it was time to create the actual function which adds the variable the user wants to create to the list and if it already exists, then modifies it. This function took 2 parameters as shown: the name of the variable and its value. Then `setenv()` was used to add/append based on the strings passed as parameters. Error checking was performed as required and 0 was returned to be able to set the EXITCODE.

```
450      /* create_shell_var() function starts here */
451      int create_shell_var(char *name, char *value) {
452
453          /* Shell var as specified by user is created */
454          /* Error checking is also performed */
455          if (setenv(name, value, 1) == -1) {
456              perror("Error when creating shell variable ");
457              return -1;
458          }
459
460          /* 0 is returned if setting of variable is successful */
461          return 0;
462      }
463  }
```



## Command-line Interface

A command-line interface had to be designed to allow the user to type through it. This was done by using the provided skeleton given in the specification. Thus, `linenoise` commands were implemented to be able to read input and free it as soon as the program is done with the command so that more commands can be entered from the user.

However, for commands to be processed by the command-line, they had to be split up into arguments to be able to be processed. Thus a command **`run()`** taking as parameters the string to be converted into arguments and the actual args array where these arguments are to be stored was created. This was built on the tokenization implementation skeleton given aswell.

Up till now, when running the program, the user is presented with a prompt to which input can be inserted and upon hitting enter, the user is displayed with a list of what he entered separated into args. This was useful to ensure that the entered input was properly tokenized into the args array.

Following this, another command called **`check_input()`** taking as parameters the split up args was created to be able to process the user input entered. With that being said, all functions to be used in this shell had to be called from within this function.

Thus, the driver code which led to the **`create_shell_vars()`** function which split up the entered args[0] (into name and value) and substituted when a \$ was detected was added to this function. This meant that now, the user could actually set a shell variable or modify an existing one.

Given that all functions being used had to be called from this **`check_input()`** function, the **`set_exitcode()`** function had to receive input to be able to modify the EXITCODE each time. This meant that each function being called had to return a specific value from **`check_input()`** so that EXITCODE is set respectively. Thus, it was decided to simply 'return' actual functions rather than simply calling them so that the value they return is immediately processed to the EXITCODE function. This was implemented for all the functions used in this assignment apart from the **`int_exit()`** function which doesn't return but simply exits the shell.

## Internal Commands

The internal commands which had to be created are the following:

- exit
- print
- chdir
- all
- source

Before starting off, certain guidelines were defined. It was already clear that all functions running on the eggshell had to be returned from **check\_input()** (except for exit). It was also important to replace a variable's name with it's corresponding value. This was required to be done so that if a user enters 'print \$USER', the actual value stored in USER is replaced with \$USER immediately. Thus this textual replacement had to be done at the very start of the **check\_input()** function.

```
654  /* check_input() function starts here */
655  int check_input(char **args) {
656
657      /* Loops through all the arguments */
658      for (int i = 1; args[i] != NULL; i++) {
659
660          /* Pointer to individual arguments */
661          char *show = args[i];
662
663          /* If a dollar sign is identified as the first char in an arg */
664          if (show[0] == '$') {
665
666              /* Its corresponding shell var is retrieved */
667              if ((show = getenv(&show[1])) == NULL) {
668                  perror("getenv() fail ");
669                  return -1;
670              }
671
672              /* If successfully retrieved, the argument is replaced with
673               * the value received back */
674              args[i] = show;
```

This was made possible by first looping through all the relevant arguments. Emphasis is put on the word 'relevant' because the first argument entered

must be a command and so there is no need to loop through it. Then, each individual argument content was added to a character pointer and the first element in it was checked if it was a \$. If so, getenv() was used to obtain the contents and replace them with what was already there.

Now that case was tackled, the escape character implementation followed. This added onto the same if statement being used to check for \$. This time, if the first element in an argument was found to be a \, this was simply removed and no substitution of variables was needed to be done.

```
676      /* If a \ is detected, this is simply removed from the argument */
677      } else if (show[0] == '\\') {
678          args[i] = &show[1];
679      }
```

Given that these guidelines were set, an if – else if – else statement had to be implemented right after to be able to return the different functions yet to be created respectively. In the if clause, the driver code for the function which created the shell variables or modified them was placed before attempting to check for internal commands. In the remaining else – if clauses, the first argument entered was compared to an internal command available and if it matches it completely, the function related to it is returned as shown:

```
747      /* The function creating the shell variables is then returned */
748      return create_shell_var(name, value);
749
750      /* else if 'exit' is detected, this section runs */
751      } else if (strcmp(args[0], "exit") == 0) {
752
753          /* exit function is called */
754          int_exit();
755
756          /* else if 'print' is detected, this section runs */
757      } else if (strcmp(args[0], "print") == 0) {
758
759          /* print function is returned */
760          return int_print(args);
761
762          /* else if 'chdir' is detected, this section runs */
763      } else if (strcmp(args[0], "chdir") == 0) {
764
765          /* chdir function is returned */
766          return int_chdir(args);
767      }
```

```
768      /* else if 'all' is detected, this section runs */
769      } else if (strcmp(args[0], "all") == 0) {
770
771          /* all function is returned */
772          return int_all();
773
774          /* else if 'source' is detected, this section runs */
775      } else if (strcmp(args[0], "source") == 0) {
776
777          /* source function is returned */
778          return int_source(args);
```

The final step in this procedure was to create all the internal commands. All the commands took in the args arguments as parameters apart from 'exit' and 'all' which did not really need these.

Starting with the exit command, this was implemented in the **int\_exit()** function. It simply displayed a 'Goodbye' message to the user and called `exit(0)` as shown:

```
466      /* int_exit() function starts here */
467      void int_exit() {
468
469          printf("Goodbye!\n\n");
470
471          /* Exits the eggshell */
472          exit(0);
473      }
```

Moving on, the print command was implemented next in the **int\_print()** function. This simply checks if more arguments apart from the first were entered. If so, the args are printed one after the other back to the user on the same line and then a new line is printed followed by a return of 0. If not, the user is informed with an appropriate error message and the function returns -1. Given that the textual swapping of variables and the escape character were handled at the start of the **check\_input()** function, nothing else needs to be done.

```
476  /* int_print() function starts here */
477  ↩ int int_print(char **args) {
478
479      /* If the user entered a second argument, this part runs */
480      if (args[1] != NULL) {
481
482          /* Arguments entered are then printed to the console */
483          for (int i = 1; args[i] != NULL; i++) {
484
485              printf("%s ", args[i]);
486
487          }
488
489          /* If not, user is informed and -1 is returned */
490      } else {
491          perror("Nothing to print ");
492          return -1;
493      }
494
495      /* A new line is added */
496      putchar('\n');
497
498      /* 0 returns if successful */
499      return 0;
500
501  }
```

Next, the **int\_chdir()** function was created. This first checks if an argument other than the first was entered. If not the user is informed of this absence and -1 is returned. If more than 1 arguments are present, the chdir() function defined in unistd.h is used to change the directory followed by modifying the CWD shell variable using getcwd() to get the current working directory and setenv() respectively. If all of this is successful, 0 is returned back to main. If not, error checking was performed to trap the error and inform the user.

```
505 int int_chdir(char **args) {
506
507     /* A character array of 100 chars is initialised */
508     char len[100];
509
510     /* If user enters more than 1 argument, this section runs */
511     if (args[1] != NULL) {
512
513         /* Directory is then changed and error checking is performed */
514         if (chdir(args[1]) == -1) {
515             perror("Error when changing directory ");
516             return -1;
517         }
518
519         /* CWD shell variable is updated with the changes */
520         if (setenv("CWD", getcwd(len, 100), 1) == -1) {
521             perror("Error when modifying directory ");
522             return -1;
523         }
524
525         /* If not, user is informed and program returns -1 */
526     } else {
527         perror("No directory specified ");
528         return -1;
529     }
530
531     /* 0 is returned if successful */
532     return 0;
```

Moving on to the 'all' command implemented in **int\_all()**, this simply makes use of the global environ variable to print out all the new shell variables and all the previously already existing environmental variables. This function then returns a value of -125 when successful. The reason why this was done was to preserve the state of EXITCODE when calling this function. When this function returns -125, EXITCODE is neither set to 0, nor to errno but is kept as it is. This comes in handy as the user is sure to view the EXITCODE of just the previous command when using this function and not having the EXITCODE reset due to a use of another command. In reality, when the user types 'print \$EXITCODE', the exit code of the previous command is printed out. However, by the time the prompt is displayed again for the user to enter another command, the exit code changes to the exit status of this print command and so is not preserved for another command to make use of it.

```
537      /* int_all() function starts here */
538      ↩ int int_all() {
539
540          /* Synopsis required to access the global environ variable */
541          extern char **environ;
542
543          /* Loops through all env variable and prints them */
544          for (int i = 0; environ[i] != NULL; i++) {
545
546              printf("%s \n", environ[i]);
547
548          }
549
550          /* Returns -125 back to main if successful */
551          return -125;
552
553      }
```

The last command 'source' is made to work in the **int\_source()** function. In it, it is first checked if more than one argument is entered, then if the file supplied is able to be opened in read mode.

```
556      /* int_source() function starts here */
557      ↩ int int_source(char **args) {
558
559          /* A file pointer is created */
560          FILE *fp;
561
562          /* A string of 1000 chars is initialised */
563          char str[1000];
564
565          /* If the user enters a second argument, this section runs */
566          if (args[1] != NULL) {
567
568              /* If file exists and is opened successfully for reading, this section runs */
569              if ((fp = fopen(args[1], "r")) != NULL) {
570
```

This follows with a while loop which keeps reading line by line until EOF is reached. The separate lines are added to a string and the last new line character is replaced with a null terminated string character.

```
571          /* Gets each line until EOF is reached */
572          while (fgets(str, 1000, fp) != NULL) {
573
574              /* Stored length of each line */
575              int length = strlen(str);
576
577              /* Replaces final new line char with null terminated string char */
578              if (str[length - 1] == '\n') {
579
580                  str[length - 1] = '\0';
581
582              }
```

The line being considered is then printed out on the terminal and the **run()** and **check\_input()** functions are called with the appropriate parameters to process the text.

```
584      /* The prompt and text from file are printed to stdout */
585      fprintf(stdout, "%s", getenv("PROMPT"));
586      fputs(str, stdout);
587
588      /* New line is placed */
589      putchar('\n');
590
591      /* The line retrieved from file is split into args */
592      run(args, str);
593
594      /* The args are then processed for output */
595      check_input(args);
```

Upon successful completion, 0 is then returned and the prompt is displayed to the user once again. Error checking was also performed in each stage so that a user can identify what was done wrong if an error pops up.

Now that all these commands were implemented and work successfully, it was time to move over to external commands.



## External Commands

Before considering how to implement the external commands, the long if statement implemented earlier for all the internal commands in the **check\_input()** function was modified. To its end, an else branch was added as the last modification required to this statement in which a function implementing these external commands would be returned. This meant that upon receiving input and splitting into args, the program checks each possible if clause for a matching command and finally resorts to executing the else branch as the last option. A return value of -1 was also added at the very last part of the **check\_input()** function, just in case for some reason none of the commands get returned back to main to set the exit code.

```
774     /* else if 'source' is detected, this section runs */
775     } else if (strcmp(args[0], "source") == 0) {
776
777         /* source function is returned */
778         return int_source(args);
779
780         /* else, the external commands function is returned */
781     } else {
782
783         return external(args);
784
785     }
786
787     /* -1 is returned on unsuccessful completion */
788     return -1;
789
790 }
791 }
```

Now that this was cleared out of the way, it was time to create the **external()** function for external commands. But before this was done, another function for the exit code called **set\_exit\_when\_fail()** was created to be used in this function. The reason why this was done will be understood in a bit but for now, the contents of this function will be explained. Like in the previous **set\_exitcode()** function, a temp character array is created to store the value of errno into a string. This string is then passed as one of the parameters in the setenv() function to set the new EXITCODE value. If this is not set successfully, error checking was performed to let the user know.

```
435  /* set_exit when fail() function starts here */
436  void set_exit_when_fail() {
437
438      /* errno value is stored in a temp string */
439      char temp[10];
440      sprintf(temp, "%d", errno);
441
442      /* EXITCODE is then set to the value of errno and error checking is done */
443      if(setenv("EXITCODE", temp, 1) == -1){
444          perror("EXITCODE not set");
445      }
446
447  }
```

Finally, the **external()** function could be created with all the prerequisites already set. In this function, the current running process is forked. The fork return value is stored in the pid variable of type pid\_t.

```
794  /* external() function starts here */
795  int external(char **args) {
796
797      /* Variable to store the exit status */
798      int status;
799
800      /* The current process running is forked */
801      pid_t pid = fork();
```

Next, an if statement was constructed to be able to control what the child does, what the parent does and to also check for errors. Given that the child always has a process ID of 0, the first if branch checked for this condition. This was done to enter the child process. Following this, the `execvp()` system call is used to branch off into the kernel and make the child execute a specific task signified by `args[0]`. If whatever the user entered exists as an external command, then the command executes, but if not, an error is displayed to the user and the child process needs to exit its execution. For the child process to exit it, the function cannot simply return -1 as the child process requires a special way for it to be exited. Thus instead of return 'EXIT\_FAILURE' was used to signify an exit due to a failed attempt. But this mode of exiting does not return anything back to the function required to set the exit code. Therefore, it is for this reason the new function **set\_exit\_when\_fail()** was created. This function is then called just before exiting the failed attempt in the child as can be seen:

```
803      /* If the child, then run this code */
804      if (pid == 0) {
805
806          /* execvp searches for a path which corresponds to the first
807           * argument entered and tries to execute the command
808           * If not, an error message is displayed and execution exits*/
809          if (execvp(args[0], args) == -1) {
810              perror("Error ");
811
812              /* function which sets exitcode to errno is called */
813              set_exit_when_fail();
814
815              /* Execution is exited with a fail */
816              exit(EXIT_FAILURE);
817          }
```

As already mentioned, error checking is performed to ensure forking was done correctly. Thus the case in which the `pid < 0` was included in the if statement informing the user with the failed forking attempt.

```
819          /* If pid has a value less than 0 forking failed */
820      } else if (pid < 0) {
821
822          perror("fork() failed");
823
824          /* function which sets exitcode to errno is called */
825          set_exit_when_fail();
826
827          /* Execution is exited with a fail */
828          exit(EXIT_FAILURE);
829      }
```

Once again, the **set\_exit\_when\_fail()** function was called to set the exit code before 'EXIT\_FAILURE' is performed. Moving on, if everything is successful, execution returns to the parent which has a process ID > 0. However, given that the parent and the child are 2 separate processes, to reduce the risk of errors, the parent process is made to wait for the child to finish executing before doing anything else. This is done to ensure that the child process does not turn into a zombie process. This is done with the `wait()` command and once finished, 0 is returned back to the main.

```
833      /* Parent wait for child to finish executing */
834      waitpid(pid, &status, 0);
835
836      /* On successful execution, 0 is returned */
837      return 0;
838  }
```

## Redirection

Now that both the internal and external commands are working, it is time to move onto redirection. In redirection, it was required that the **check\_input()** function was able to identify the different redirection operators and return the appropriate functions to complement them. This was made possible by declaring 2 global arrays. An `*ops[]` array and an another array of function pointers called `*ops_address[]` to store the addresses of the respective function pointers. The first one was populated with the actual operators to look out for when searching for redirect operators whilst the second contained the memory addresses which pointed to the respective functions. These can be seen below:

```
199      /*                                GLOBAL SCOPE VARIABLES / ARRAYS                                */
200
201
202      /* A global array of all the redirect operators implemented */
203      char *ops[] = {
204          ">",
205          ">>",
206          "<",
207          "<<<"
208      };
209
210      /* A global array of function pointers which point to the
211      * corresponding functions based on the redirect operator entered */
212      int (*ops_address[])(char **, int) = {
213          &forward_1_arrow,
214          &forward_2_arrows,
215          &back_1_arrow,
216          &back_3_arrows
217      };
218
```

Now that the required global variables were set, it was time to write code which identifies these operators in the **check\_input()** function. This was made possible by including the code at the start of the function in the for loop which loops through all the relevant arguments, right after the check for escape characters statement. Once again, emphasis is put on 'relevant' because it does not make sense to place a redirect operator as the first

argument. This is because redirect operators must have text/commands on both sides of the operator to be able to work.

The way these redirect operators were checked was through a for loop within the bigger for loop which loops through the relevant args. This for loop was made to loop 4 times, each time comparing the argument at position *i* with an element in the ops array. If a match is found, the array of function pointers is returned back and the corresponding function is run.

```
681      /* A for loop which loops through all redirection operators */
682      for (int j = 0; j < 4; j++) {
683
684          /* If an entered argument matches a redirection operator
685           * in the ops array, then the corresponding function pointer
686           * value stored in that particular position is returned
687           * which stores the address of the function to be computed*/
688          if (strcmp(args[i], ops[j]) == 0) {
689              return (*ops_address[j])(args, i);
690          }
691      }
692  }
693  }
694  }
695  }
```

Given that the prerequisites were lined out for all the possible redirect operator cases, it was time to implement the separate redirect functions. It was taken as a convention that each of the 4 functions had the arguments passed as parameters, together with the position of the redirect operator in the args.

Starting with the '**>**' redirect operator, which was coded in the **forward\_1\_arrow()** function, a file descriptor was created. Then, the **open()** system call was used to create a file based on the name entered by the user in **args[ i + 1]**. The file created was passed the flags shown below to be able to be written to only while at the same giving it the ability to be created, if it does not exist and to be truncated if it already exists. Finally the 0644 permission was given to the file to be able to open it directly and view its contents from outside the shell. Error checking was also performed to ensure that everything works well.

```

850      /* A file is opened in write mode with the appropriate flags set */
851      /* If the file desc is not set accordingly, open() failed and -1 is returned */
852      if ((fd = open(args[i + 1], O_WRONLY | O_CREAT | O_TRUNC, 0644)) == -1) {
853          perror("open() fail");
854          return -1;
855      }

```

Next, the commands entered before the '>' operator, meaning the commands whose output is to be stored in the file are added to a newline string.

```

857      /* All arguments entered up till the > operator are added to an array */
858      for (a = 0; a < i; a++) {
859          strcat(newline, args[a]);
860          strcat(newline, " ");
861      }

```

Following this, the normal STDOUT and STDERR states are stored to 2 separate file descriptors. Then the output and error streams are redirected to the file by using dup2() on the open file's file descriptor. This was done so that anything processed after this point is outputted in the file.

```

863      /* Temporarily stores the STDOUT state to an int variable */
864      if ((save_stdout = dup(STDOUT_FILENO)) == -1) {
865          perror("dup() failed");
866          close(fd);
867          return -1;
868      }
869
870      /* Temporarily stores the STDERR state to an int variable */
871      if ((save_stderr = dup(STDERR_FILENO)) == -1) {
872          perror("dup() failed");
873          close(fd);
874          return -1;
875      }
876
877      /* The file descriptor opened is set to STDOUT */
878      if ((dup2(fd, 1)) == -1) {
879          perror("dup2() failed");
880          close(fd);
881          return -1;
882      }
883
884      /* It is also set to STDERR */
885      if ((dup2(fd, 2)) == -1) {
886          perror("dup2() failed");
887          close(fd);
888          return -1;
889      }

```

The file descriptor used is then closed as it is useless after this point. The commands stored in the newline string are then split again into args through the **run()** function and processed using the **check\_input()** function. Their output is then automatically redirected to the file as requested by the user.

```
891      /* The file descriptor is then closed */
892      close(fd);
893
894      /* The newline string is then separated into args */
895      run(args, newline);
896
897      /* The args are processed */
898      check_input(args);
899
```

The final step required was to restore the state of the STDIN and STDERR streams as failure to do so will result in all output being redirected to the file created even after the function returns.

```
900      /* The STDOUT is then restored to the usual stdout */
901      if ((dup2(save_stdout, STDOUT_FILENO)) == -1) {
902          perror("dup2() failed");
903          return -1;
904      }
905
906      /* The STDERR is then restored to the usual stderr */
907      if ((dup2(save_stderr, STDERR_FILENO)) == -1) {
908          perror("dup2() failed");
909          return -1;
910      }
911
912      /* 0 is then returned if everything is successful */
913      return 0;
914
915 }
```

Following successful operation, 0 is returned back to the main function.

Moving on, the '>>' redirect operator was implemented next in the **forward\_2\_arrows()** function. Its purpose is very similar to the '>' redirect operator with the only difference being that any form of output returned from a function is appended to a new / existing file rather than truncating the file each time. With that being said, this function is implemented exactly the same as the previous one with the only difference being the flags used when opening the file.

When the open() system call is used, the write only and create file if it does not exist flags were passed to it as before. However, the O\_TRUNC flag which simply overwrites the previous contents in the file is not included. This time, the O\_APPEND flag is inserted which on simple terms appends output to the file rather than overwriting it.

```
927      /* A file is opened in write and append mode with the appropriate flags set */
928      /* If the file desc is not set accordingly, open() failed and -1 is returned */
929      if ((fd = open(args[i + 1], O_WRONLY | O_CREAT | O_APPEND, 0644)) == -1) {
930          perror("open() fail");
931          return -1;
932      }
```

As explained, the remainder of the operations done for this redirection mode are the same and so they won't be explained again.



The '`<`' redirection operator was implemented next in the **back\_1\_arrow()** function. In this function, a file was opened using the `open()` system call which opens a file (if it exists) based on what the user entered. The only flag passed into this function is the `O_RDONLY` as the file's contents are only required to be read. Also, the permission of `0444` was granted to the file so that it can be read from outside the shell.

```
1004      /* A file is opened in read only mode with the appropriate flags set */
1005      /* If the file desc is not set accordingly, open() failed and -1 is returned */
1006      if ((fd = open(args[i + 1], O_RDONLY, 0444)) == -1) {
1007          perror("open() fail");
1008          return -1;
1009      }
```

Once again, the arguments entered before the redirection operator are added to a created string to be able to be accessed at a later stage.

```
1011      /* All arguments entered up till the < operator are added to an array */
1012      for (a = 0; a < i; a++) {
1013          strcat(newline, args[a]);
1014          strcat(newline, " ");
1015      }
```

Next, the current `STDIN` state is saved to a file descriptor to be able to be reset later. Following this, the opened file's file descriptor is then set to be the current `STDIN`. This meant that any functions processed from now on would receive input from what is already present in the file.

```
1017      /* Temporarily stores the STDIN state to an int variable */
1018      if ((save_stdin = dup(STDIN_FILENO)) == -1) {
1019          perror("dup() failed");
1020          close(fd);
1021          return -1;
1022      }
1023
1024      /* The file descriptor opened is set to STDIN */
1025      if ((dup2(fd, 0)) == -1) {
1026          perror("dup2() failed");
1027          close(fd);
1028          return -1;
1029      }
```

Given that this operation of changing the state was successful, the contents of the file are added to the existing string and then they are separated into args and passed to the **check\_input()** function.

```
1031      /* Loops through all the remaining args and appends them to the newline string */
1032      for (a = i + 2; args[a] != NULL; a++) {
1033          strcat(newline, args[a]);
1034          strcat(newline, " ");
1035      }
1036
1037      /* The newline string is then separated into args */
1038      run(args, newline);
1039
1040      /* The args are processed */
1041      check_input(args);
```

Output is then shown on the console if the operation is successful. To close off, the STDIN is then refactored back to it's original state and the file descriptor of the file being used is closed just before the program returns 0 back to main to indicate a successful implementation.

```
1043      /* The STDIN is then restored to the usual stdin */
1044      if ((dup2(save_stdin, STDIN_FILENO)) == -1) {
1045          perror("dup2() failed");
1046          return -1;
1047      }
1048
1049      /* The file descriptor is then closed */
1050      close(fd);
1051
1052      /* 0 is then returned if everything is successful */
1053      return 0;
1054  }
```

The final redirection operator was implemented differently. The program was written so as to accept this operator as so: "<<<". This was done because this operator made use of a here string for redirecting the input written by the user. The user can enter any number of lines he wants to but must terminate the read in of information by closing the inverted comma. The implementation of this will now be explained.

In the **back\_3\_arrows()** function, 4 arrays of characters were initialized at the start. Next, all the arguments entered up till the '<<<' operator were added to one of these arrays for to be used later.

```

1065     char newline[1200] = "";
1066     char finalline[500] = "";
1067     char temp_array1[500] = "";
1068     char temp_array2[500] = "";
1069
1070     char c;
1071
1072     /* All arguments entered up till the '<<<' operator are added to an array */
1073     for (a = 0; a < i; a++) {
1074         strcat(finalline, args[a]);
1075         strcat(finalline, " ");
1076     }

```

Following this, the '<' redirect operator is appended to the string in finalline together with a file name which will temporarily store the output. The reason for this will be understood soon.

Next, the word print is concatenated to the newline string followed by the here string written on the same line as the operator.

```

1078     /* Text is appended to the finalline string */
1079     strcat(finalline, "< temp.txt");
1080
1081     /* Text is added to the newline string */
1082     strcat(newline, "print ");
1083
1084     /* The arguments after the '<<<' operator and on the same line as it
1085     * are added to the newline string */
1086     for(a = i+1 ; args[a] != NULL ; a++){
1087         strcat(newline, args[a]);
1088         strcat(newline, " ");
1089     }

```

To finally start making changes visible, some more code was added as can be seen below to pass the stdout to the temp.txt file. This was done by

making use of the '>' redirect operator to redirect the output given from printing the first line of here strings entered to a given file. Following this, the newline string is split into args and processed and then reset using the `memset()` function.

```
1091      /* More text is added to the newline string */
1092      strcat(newline, "> temp.txt");
1093
1094      /* The newline string is split into args */
1095      run(args, newline);
1096
1097      /* These args are then processed */
1098      check_input(args);
1099
1100      /* The character string newline is then emptied */
1101      memset(&newline[0], 0, sizeof(newline));
```

When this is done, the file `temp.txt` is created (if it does not exist) and the contents are added to the file. Following this, a while loop keeps listening for user input until a ' character is entered. All the contents entered are then appended to an array which sets the last character to a '\0'.

```
1103      /* Keeps reading the here string until user hits ' */
1104      while((c = getchar()) != '\n'){
1105
1106          /* Appends each entered character to another array */
1107          temp_array1[b++] = c;
1108      }
1109      /* Sets last element to '\0' */
1110      temp_array1[b] = '\0';
```

Then another while loop was implemented to read all the other separate lines entered by the user and individually append them to other lists to be finally split into args and executed. This means that the individual lines entered in the here string are appended to the contents already found in the `temp.txt` file.

```

1116      /* Loops until '\0' is reached */
1117      while(temp_array1[a] != '\0'){
1118
1119          /* If '\n' is detected, certain contents are appended to newline string */
1120          if(temp_array1[a] == '\n'){
1121              strcat(newline, "print ");
1122              strcat(newline, temp_array2);
1123              strcat(newline, " >> temp.txt");
1124
1125              /* These are then split into args */
1126              run(args, newline);
1127
1128              /* The args are then evaluated */
1129              check_input(args);
1130
1131              /* a increments and b is 0 */
1132              a++;
1133              b = 0;
1134
1135              /* The character strings newline & temp_array2 are then emptied */
1136              memset(&temp_array2[0], 0, sizeof(temp_array2));
1137              memset(&newline[0], 0, sizeof(newline));
1138
1139              /* continues */
1140              continue;
1141          }
1142
1143          /* a char in temp_array1 is added to temp_array2 */
1144          temp_array2[b++] = temp_array1[a++];
1145      }

```

Nearing towards the end, another set of strcat() functions were used to append the last line of the here string entered to a command which appends the entered text to the temp.txt file. From this point on, all the here strings entered by the user have been added to the temp.txt file.

```

1147      /* More contents are added to newline string */
1148      strcat(newline, "print ");
1149      strcat(newline, temp_array2);
1150      strcat(newline, " >> temp.txt");
1151
1152      /* These are then split into args */
1153      run(args, newline);
1154
1155      /* And evaluated for output */
1156      check_input(args);
1157
1158      printf("---\n");

```

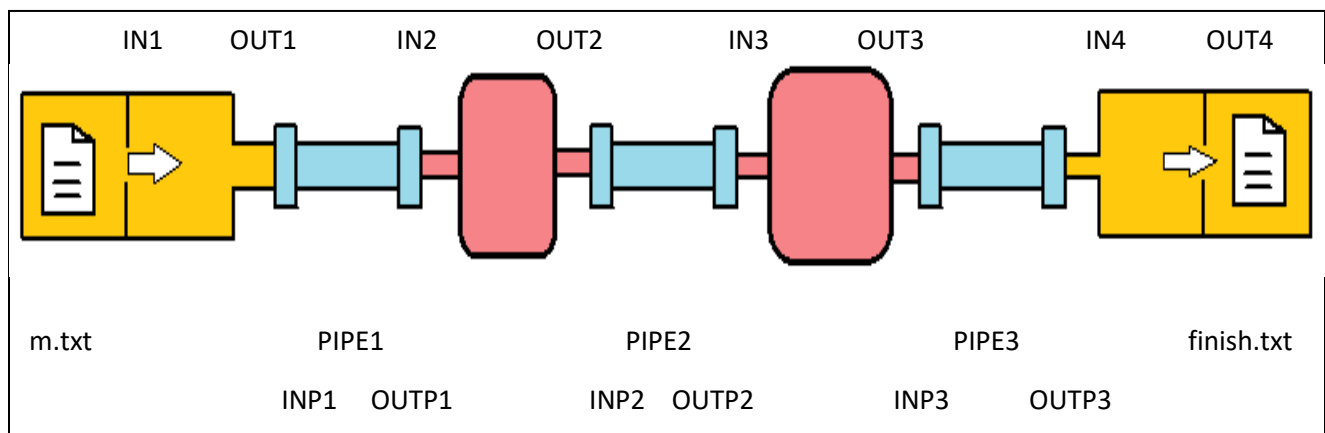
It was now time to actually pass the contents in the temp.txt file to the command entered by the user. For this to be done, the '<' redirection operator appended earlier to the finalline string is used to redirect the contents of the file into the function entered by the user. Thus, all that was required to be done at this point was to split the finalline string into args and

call the **check\_input()** function to evaluate the finalline string. If everything is successful, 0 is then returned back to the main function.

```
1160      /* The finalline string is split into args */
1161      run(args, finalline);
1162
1163      /* The input is evaluated */
1164      check_input(args);
1165
1166      /* 0 is returned on success */
1167      return 0;
1168
1169 }
```

# Piping

When trying to implement piping for just 1 pipe, this was managed in no time. However, converting the implementation to accept multiple pipes was hard to achieve and took a lot of thinking. Understanding how the previous implementation could be stretched to cover both single and multiple pipes was done with the help of the following diagram:



In the diagram I drew, one can see the piping process for 4 commands with 3 pipes in between. Consider the piping process:

```
cat m.txt      |      grep a      |      wc -l      |      figlet > finish.txt
```

This process opens a file called m.txt, selects the contents containing the letter 'a' only then counts all the lines that the grep process has selected, pipes the number into figlet and redirects the figlet output to a new file called finish.txt.

Breaking down the process bit by bit, the IN1 (input) of the m.txt file is passed into the cat command. This produces OUT1 (output) which basically has all the contents in the file. The tricky part starts at this point when the OUT1 needs to be redirected into INP1 (input of pipe 1). This is then passed to

OUTP1 (output of file 1). OUTP1 is then passed as input to the grep command, ie. into IN2 where it is evaluated and an output is produced. The produced output in OUT2 is then redirected to INP2 and then to OUTP2 from where it is passed to the next command as input. This process is repeated again and again until no more pipes are remaining. The final command needs to then redirect it's output to the file given and so, it's STDOUT must be made to point to the file's particular file descriptor. If there was no output redirection at the end, meaning that figlet was the last command, then STDOUT simply is redirect back to it's original state and output is shown on the console.

Now that this procedure for multiple pipes was understood, it was time to start writing the C code.

To start out, the following code was added in the **check\_input()** function:

```
697      /* Variable j is initialised to 0 */
698      int j = 0;
699
700      /* Section of code which checks for pipes */
701      for (int i = 1; args[i] != NULL; i++) {
702
703          if (strcmp(args[i], "|") == 0) {
704
705              /* If found, the variable j holding the number of pipes is incremented */
706              j++;
707          }
708      }
709
710      /* If more than one pipe */
711      if (j != 0) {
712
713          /* Piping function is returned */
714          return piping(args, j);
715      }
716
717  }
```

This code was added to check if any pipes were present before attempting to check for any internal or external commands. It simply incremented the variable j based on the number of pipes found. If j is found to not be equal to 0, then the **piping()** function is returned with the args and pipecount passed as parameters.



Now that these prerequisites were set, it was time to start implementing the actual function. In this function, some variables were initialized at the start. Some worth mentioning are:

- An array of commands which would store the separate commands entered separated by the | operator.
- Another array which stores pipe file descriptors for each existing pipe so that these can be accessed individually and set accordingly
- 2 int variables initially set to 0 but which would be able to store the STDIN and STDOUT state

Moving on, the first operation performed was that of appending all the entered arguments to a string called str1.

```

1194     /* Loops through all entered args */
1195     for (int a = 0; args[a] != NULL; a++) {
1196
1197         /* Appends args to the str1 char string */
1198         strcat(str1, args[a]);
1199         strcat(str1, " ");
1200     }

```

The next thing done was the splitting up of this string based on the pipe operators. Any extra spaces before or after the commands were removed before adding these separate commands to the commands array.

```

1202     /* Sets the 1st token to delimit "|" */
1203     token = strtok(str1, "|");
1204
1205     /* Loops through all tokens */
1206     while (token != NULL) {
1207
1208         /* Populates commands array with strings delimited */
1209         commands[tokenIndex] = token;
1210
1211         /* Removes the first blank space */
1212         if (commands[tokenIndex][0] == ' ') {
1213             commands[tokenIndex] = &commands[tokenIndex][1];
1214         }
1215
1216         /* Gets length of a command */
1217         int length = strlen(commands[tokenIndex]);
1218
1219         /* Removes the last white space */
1220         if (commands[tokenIndex][length - 1] == ' ') {
1221             commands[tokenIndex][length - 1] = 0;
1222         }
1223
1224         /* Constructs required to complete the looping process */
1225         token = strtok(NULL, "|");
1226         tokenIndex++;

```

The usual STDIN and STDOUT states were set to the variables previously set to 0. This was done so that the standard states could be reset at the end of the piping procedure.

```
1230      /* Temporarily stores the STDIN state to an int variable */
1231      if ((save_stdin = dup(STDIN_FILENO)) == -1) {
1232          perror("dup() failed");
1233          return -1;
1234      }
1235
1236      /* Temporarily stores the STDOUT state to an int variable */
1237      if ((save_stdout = dup(STDOUT_FILENO)) == -1) {
1238          perror("dup() failed");
1239          return -1;
1240      }
```

Next, all the required pipes were created to be able to redirect any input / output through the file descriptors coming with it.

```
1242      /* Loops and creates all required pipes */
1243      for (int a = 0; a < pipecount + 1; ++a) {
1244
1245          if (a < pipecount) {
1246
1247              /* Validation is performed if creating a particular pipe is unsuccessful */
1248              if ((pipe(pipes[a]) < 0)) {
1249                  perror("pipe() failed");
1250                  return -1;
1251              }
1252          }
```

Now that the pipes were created, it was time to set the pipe's file descriptors to read and write to and from commands respectively. This was one of the toughest situations encountered in all of the assignment. However, after revising the piping system previously drawn, it was clear that in the stages where we have both consumers and producers, the file descriptor of the previous pipe has to be set to the active read mode, whilst that of the current one was required to be set to write mode. With that said, the previous pipe was set to receive STDIN, whilst the current one was set to write to STDOUT.

The producer-consumer pair was understood and was ready to be implemented but before doing so, the cases in which only a producer exists or only a consumer exists had to be taken in full consideration. These cases referred to the first and last commands respectively. Taking a look at the

diagram drawn up once again, one can see that the first process requires rewiring of its stdout to the input of a pipe whilst the last command requires only to read input from a pipe and process it to the standard stdout of the terminal. Thus, all that has already been described above was achieved by adding the following code:

```

1254      /* If not the first command, run this */
1255      if (a != 0) {
1256
1257          /* Sets the file descriptor of the previous pipe to receive STDIN */
1258          if ((dup2(pipes[a-1][0], STDIN_FILENO)) < 0) {
1259              perror("Changing STDIN failed");
1260              return -1;
1261          }
1262      }
1263
1264
1265      /* If not the last command, run this */
1266      if (a < pipecount) {
1267
1268          /* Sets the file descriptor of the current pipe to write to STDOUT */
1269          if ((dup2(pipes[a][1], STDOUT_FILENO)) < 0) {
1270              perror("Changing STDOUT failed");
1271              return -1;
1272          }

```

An else statement was then added to the last if statement created to restore the final STDOUT back to the terminal so that any possible output after piping finishes is visible on the screen. This can be seen below:

```

1274      /* If the last command is reached, this section runs */
1275      }else{
1276
1277          /* The STDOUT is set once again to the normal stdout meaning
1278           * that the final result is printed out to the terminal*/
1279          if ((dup2(save_stdout, STDOUT_FILENO)) < 0) {
1280              perror("Changing STDOUT to terminal failed");
1281              return -1;
1282          }
1283      }
1284

```

After having set the pipe file descriptor to the current respective read or write as required, this file descriptor was closed as it was not needed anymore and also to avoid any unnecessary errors.

```

1286      /* If not the last command, run this */
1287      if(a < pipecount){
1288          /* Closes the current write pipe file descriptor */
1289          if(close(pipes[a][1]) < 0){
1290              perror("close() failed");
1291              return -1;
1292          }
1293      }
1294
1295      /* If not the first command, run this */
1296      if(a > 0){
1297          /* Closes the previous read pipe file descriptor */
1298          if(close(pipes[a - 1][0]) < 0){
1299              perror("close() failed");
1300              return -1;
1301          }
1302      }
1303  }

```

Finally, the separate commands stored in the commands 2d array were individually split into args and processed. This was done for each command from one iteration to the next.

```

1305      /* The commands are then tokenised and split into args */
1306      run(args, commands[a]);
1307
1308      /* The commands are then processed */
1309      check_input(args);
1310  }

```

To close off this piping method, the STDIN, which at this point is still set to read input from the last pipe is restored the usual standard input so that the user is able to write input to the shell once again. Then, the file descriptors used to store the state of the standard stdin and stdout are closed as they are no longer required and the program returns 0 to main.

```

1312      /* The STDIN is then restored to the usual stdin */
1313      if((dup2(save_stdin, STDIN_FILENO)) < 0){
1314          perror("dup2() failed");
1315          return -1;
1316      }
1317
1318      /* The previously duped file descriptors are also closed */
1319      if((close(save_stdin) == -1) || (close(save_stdout) < 0)){
1320          perror("close() failed");
1321          return -1;
1322      }
1323
1324      /* 0 is returned on successful implementation */
1325      return 0;
1326  }

```

## Process Management

When dealing with this section, it was required that the SIGINT signal triggered when pressing 'Ctrl + C' would be trapped and forwarded to the process currently running. For this to be done, the following function was created to flush the standard out when such a signal was detected:

```
1329 /* sigint handler() function starts here */
1330 void sigint_handler(int signal) {
1331
1332     /* STDOUT is flushed */
1333     fflush(stdout);
1334
1335     /* New line character is printed */
1336     putchar('\n');
1337 }
```

This function was then required to be called from main as follows to be able to make it work:

```
254 /* A call to the function handling the SIGINT signal */
255 signal(SIGINT, sigint_handler);
```

However, an infinite while loop had to be added over the linenoise function which reads input from user and stores it in a character pointer up till the point where the EXITCODE function is called to set this shell variable. This also implied changes to how linenoise is read. Previously, linenoise would keep running until NULL is returned. If so, the program quits. However, a NULL would be returned whenever CTRL C would be pressed, therefore, the linenoise function for getting the line was simply removed from the while loop which ensured that NULL was not returned and placed in the infinite while loop just discussed. Following this an if statement for NULL was created so that the user is shown that CTRL-C was pressed and the line entered is freed and the program continues execution. These changes explained can be seen below:

```
254      /* A call to the function handling the SIGINT signal */
255      signal(SIGINT, sigint_handler);
256
257      /* A while loop which provides the user with a PROMPT for
258       * entering commands. This keeps listening until the user
259       * enters text. When text is entered, it is processed*/
260      while (true) {
261
262          /* Text entered by user is stored in line */
263          line = linenoise(getenv("PROMPT"));
264
265          /* Handles the case in which null is returned */
266          if (line == NULL) {
267              printf("Ctrl-C\n");
268              linenoiseFree(line);
269              continue;
270          }
271
272          /* Handles the case in which user hits 'Enter' without any commands */
273          if (strlen(line) == 0) {
274              continue;
275          }
276
277          /* A linenoise function which appends an inserted
278           * command to the list of inserted commands */
279          linenoiseHistoryAdd(line);
280
281          /* A function which separates the inserted string
282           * entered by the user into arguments and stores
283           * them in an arguments array */
284          run(args, line);
285
286          /* These arguments are then passed to a function which
287           * identifies the command entered and the return value
288           * returned is stored in the placeholder return_val */
289          return_val = check_input(args);
290
291          /* The EXITCODE is set based on the value returned */
292          set_exitcode(return_val);
293      }
294
295      /* The previously allocated memory for the line is now freed */
296      linenoiseFree(line);
297
298
299      /* Return value of main function */
300      return 0;
301  }
```

## Testing

In this section, the whole program will be tested to show the correct functioning of this program.

Starting out, when compiling and running the program on the command line, the following prompt is displayed to the user:

```
tristan_oa@Oas-VirtualMachine:~/Desktop/os-assginment-eggshell$ gcc eggshell.c l
inenoise.c -o e
tristan_oa@Oas-VirtualMachine:~/Desktop/os-assginment-eggshell$ ./e
>>>
```

To kick off, the shell was first tested for it's error checking ability. A simple test to check the endurance of the shell was to constantly hit 'ENTER' and see what happens:

[illegible]

As can be seen, this was handled correctly by the shell as the prompt kept on being returned without any problems.

Next, some garbage input was entered to test what the program would output and as expected, an error message is displayed to the user:

```
>>> How does this work?
Error : No such file or directory
>>> What?
Error : No such file or directory
>>> 123457890
Error : No such file or directory
>>>
```

Moving on to the first major part in this shell, the shell variables were required to be tested. Their values were checked for by simply printing out the separate values:

```
>>> print $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
>>> print $PROMPT
>>>
>>> print $PWD
/home/tristan_oa/Desktop/os-assgiment-eggshell
>>> print $USER
tristan_oa
>>> print $HOME
/home/tristan_oa
>>> print $SHELL
/bin/bash
>>> print $TERMINAL
/dev/pts/0
>>> print $EXITCODE
0
>>> █
```

This shows that the shell variables unique to the eggshell had been properly populated during startup. It was also evident that other variables like PATH and SHELL were inherited from the previous process to this current one.

Testing next is the ability for the user to create shell variables or modify existing ones. This example shows a shell variable being modified and another one being created.



```
>>> PROMPT
Error : No such file or directory
>>> PROMPT=
No value entered: No such file or directory
>>> PROMPT=Write:
Write:
Write:TEST=
No value entered: No such file or directory
Write:TEST
Error : No such file or directory
Write:TEST=UBUNTU
Write:print $UBUNTU
getenv() fail : No such file or directory
Write:print TEST
TEST
Write:print $TEST
UBUNTU
Write:
```

As can be seen in the last example done, a shell variable TEST was created. But before, some error checking was performed. After successfully setting the variable TEST to UBUNTU, the value of UBUNTU was asked for but since, it was referenced the way it was and not the other way round, then no value was printed out and an error was displayed as expected. Next, just the word TEST was printed out since it had no \$ in front and finally a \$ was placed and the value stored in TEST was returned.

The last test performed to shell variables was that of creating a shell variable with a value of another existing shell variable. This can be seen below:

```
>>> print $FULLNAME
getenv() fail : No such file or directory
>>> FULLNAME=$USER
>>> print $FULLNAME
tristan_oa
>>>
```

Testing out now some internal commands created.

The first to be tested was the exit command. As expected, this printed a Goodbye message and exited the shell:

```
>>> exit
Goodbye!

tristan_oa@Oas-VirtualMachine:~/Desktop/os-assginment-eggshell$
```

Next, the program was re-run and the print command was tested.

```
>>> print hello, this is a test!
hello, this is a test!
>>> print $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
>>> print \SPATH
SPATH
>>> print Hello \USER, welcome to this eggshell $USER
Hello $USER, welcome to this eggshell tristan_oa
>>>
```

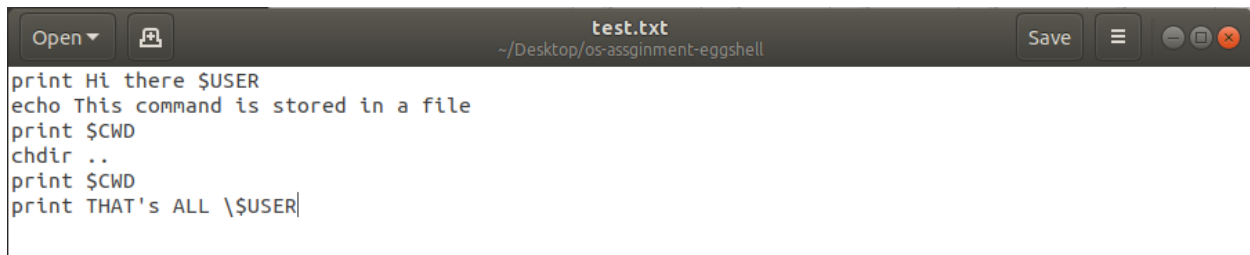
Following this, the chdir command was tested:

```
>>> print $CWD
/home/tristan_oa/Desktop/os-assginment-eggshell
>>> chdir ..
>>> print $CWD
/home/tristan_oa/Desktop
>>> chdir
No directory specified : No such file or directory
>>> chdir blablabla
Error when changing directory : No such file or directory
>>> print $CWD
/home/tristan_oa/Desktop
>>> chdir ..
>>> print $CWD
/home/tristan_oa
>>> chdir Desktop
>>> print $CWD
/home/tristan_oa/Desktop
>>>
```

Next, the all command was tested and it printed out all the environmental variables associated to the running shell:

```
LOGNAME=tristan_oa
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
XDG_RUNTIME_DIR=/run/user/1000
XAUTHORITY=/run/user/1000/gdm/Xauthority
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
LC_IDENTIFICATION=mt_MT.UTF-8
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
SESSION_MANAGER=local/Oas-VirtualMachine:@/tmp/.ICE-unix/983,unix/Oas-VirtualMachine:/tmp/.ICE-unix/983
LESSOPEN=| /usr/bin/lesspipe %s
GTK_IM_MODULE=ibus
LC_TIME=mt_MT.UTF-8
OLDPWD=/home/tristan_oa
_=./e
PROMPT=>>>
CWD=/home/tristan_oa/Desktop
TERMINAL=/dev/pts/0
EXITCODE=2
TEST=UBUNTU
FULLNAME=tristan_oa
>>> |
```

Finally, the source command was tested last. The file to be tested had the following commands written into it and was created in the same directory:



```
print Hi there $USER
echo This command is stored in a file
print $CWD
chdir ..
print $CWD
print THAT's ALL \ $USER|
```

First it was written without specifying any file name. Then, an incorrect file name was given followed by a correct one:

```
>>> source
No filename specified: No such file or directory
>>> source testing.txt
File may not exist: No such file or directory
>>> source test.txt
>>> print Hi there $USER
Hi there tristan_oa
>>> echo This command is stored in a file
This command is stored in a file
>>> print $CWD
/home/tristan_oa/Desktop/os-assginment-eggshell
>>> chdir ..
>>> print $CWD
/home/tristan_oa/Desktop
>>> print THAT's ALL \ $USER
THAT's ALL $USER
>>> █
```

The external commands were then required to be tested. Starting out, the echo command was tested followed by ls

```
>>> echo Hi there
Hi there
>>> ls
eggshell_backup os-assginment-eggshell SharedFiles
>>> chdir os-assginment-eggshell
>>> ls
a.out cmake-build-debug CMakeLists.txt debug e eggshell.c linenoise.c lin
enoise.h linenoise-master temp.txt test.txt
>>> █
```

Next, it was required to be shown that programs launched from eggshell contain egg-shell specific shell variable definitions:

```
>>> echo Textual representations of the following should be performed:
Textual representations of the following should be performed:
>>> echo $TERMINAL and also $CWD
/dev/pts/0 and also /home/tristan_oa/Desktop/os-assginment-eggshell
>>>
```

Redirection operators were tested next:

Starting out the ' $>$ ' output redirection operator was tested first. This was done by first checking the contents found in a file. Then the word ZOO was printed to the file and this was printed out to the user. As expected, all the contents were erased and ZOO replaced everything.

```
>>> cat test.txt
print Hi there $USER
echo This command is stored in a file
print $CWD
chdir ..
print $CWD
print THAT's ALL \ $USER
>>> print ZOO > test.txt
>>> cat test.txt
ZOO
>>>
```

Next the ' $>>$ ' output redirection arrow was tested and this appended more contents to this file as shown:

```
>>> cat test.txt
ZOO
>>> print orangutan >> test.txt
>>> echo leopard >> test.txt
>>> print $USER >> test.txt
>>> echo dog >> test.txt
>>> echo cat >> test.txt
>>> cat test.txt
ZOO
orangutan
leopard
tristan_oa
dog
cat
>>> 
```

Input redirection operators were also tested, starting with '<'. First, a non-existent file was passed to the sort command and an error was given as expected. Next, the contents of the test.txt file were redirected as input into the sort function and the following output was given. Finally, the test.txt file was passed as input once again but its output was redirected to a newly created file called new:

```
>>> sort < nonexistent.txt
open() fail: No such file or directory
>>> cat test.txt
ZOO
orangutan
leopard
tristan_oa
dog
cat
>>> sort < test.txt
cat
dog
leopard
orangutan
tristan_oa
ZOO
>>> cat new.txt
cat: new.txt: No such file or directory
>>> sort < test.txt > new.txt
>>> cat new.txt
cat
dog
leopard
orangutan
tristan_oa
ZOO
>>>
```

The final “<<<” redirect operator was tested. This was used to sort an entered here string as shown:

```
>>> sort <<<' work
test
sort'
```

Was then sorted into:

```
sort
test
work
>>>
```

Now that all the redirect operators were tackled, it was finally time to test piping. This was tested using 2, 3 and 4 commands as shown:

[illegible]

```
>>> cat test.txt
200
orangutan
leopard
tristan_oa
dog
cat
>>> cat test.txt | wc -l | figlet
  _
 /  \
|    |
| (  ) |
|    |
 \  _/
  _
```

```
cat: new.txt: No such file or directory
>>> cat test.txt | grep a | wc -l | figlet > new.txt
>>> cat new.txt

  _ _ _
 | | | |
 | | | | _
 | _ _ _ |
   | _ |

>>>
```

All commands worked as expected and piping was successful. Finally, the CTRL + C command was tested for when running an infinite loop of 'y' by typing yes into the command line. As can be seen, the loop then exits but the program does not even when hitting CTRL C more than once:

```
y  
y  
y  
y  
y  
y  
y  
y  
y  
y  
y  
y  
y  
y  
y  
y  
y  
y  
y  
y  
y  
y  
y  
y^C  
>>>  
Ctrl-C  
>>>  
Ctrl-C  
>>>
```

Another test done on this was that of pinging a site and using ctrl c to stop the ping:



```
>>> ping www.universityofmalta.com
PING www.universityofmalta.com (91.195.240.94) 56(84) bytes of data.
64 bytes from 91.195.240.94 (91.195.240.94): icmp_seq=1 ttl=49 time=62.0 ms
64 bytes from 91.195.240.94 (91.195.240.94): icmp_seq=2 ttl=49 time=57.7 ms
64 bytes from 91.195.240.94 (91.195.240.94): icmp_seq=3 ttl=49 time=56.5 ms
64 bytes from 91.195.240.94 (91.195.240.94): icmp_seq=4 ttl=49 time=57.3 ms
64 bytes from 91.195.240.94 (91.195.240.94): icmp_seq=5 ttl=49 time=58.0 ms
64 bytes from 91.195.240.94 (91.195.240.94): icmp_seq=6 ttl=49 time=56.9 ms
64 bytes from 91.195.240.94 (91.195.240.94): icmp_seq=7 ttl=49 time=56.6 ms
64 bytes from 91.195.240.94 (91.195.240.94): icmp_seq=8 ttl=49 time=56.6 ms
^C

--- www.universityofmalta.com ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 7011ms
rtt min/avg/max/mdev = 56.516/57.752/62.036/1.722 ms
>>>
Ctrl-C
>>>
Ctrl-C
>>>
Ctrl-C
>>>
```

In the end, ctrl-c was pressed multiple times to test if the program exits but this did not.