

IRIS NICOLE CARSON
ANTONIO RAFAEL CASTRO
JOSHUA MANUEL LOUISE KEMPIS

4 BS CpE

February 23, 2022

Project 1 Final Report

Rudimentary Shell Interpreter

Overview

A C++ program was created to simulate a basic shell interface that allows UNIX-based user commands to be inputted and processed in a virtual terminal simulation. The commands are parsed into a process depending on the operator being called: pipe ('|'), input ('<'), and output ('>'). These operators utilize built-in system calls like fork(), exec(), wait(), dup2(), and pipe() which are native to Linux and MacOS systems.

Code Collaboration Overview

Each member of the group added inputs into the structure of the final code which included the parsing function, the different implementations for each operator, and the system calls that would be used to run the different commands. A minor adjust would be made for the macOS version of the cpp file which removes the <windows.h> header and adjust the “_popen” and “_pclose” command to “popen” and “pclose” respectively.

Seeing how these headers would be possible to be removed completely from the final code, a project file was created which housed the necessary code from the previous version was made with compatibility in mind. For the final version of the code, it was made more universal working across systems such as the built-in Terminal on MacOS as well as on Cygwin in the Windows operating system.

Implementation of Functions and Operations

The functions and operations are mostly comprised of the Legacy functions and Core Functions. The former points towards functions and code that were modified based on previous outputs from the group. These include the parsing function and the fileRead() and fileWrite() functions that were present in prior outputs. Core Functions point toward the necessary functions that make the program work the way it is designed to. These core functions operate as what the rudimentary shell program requires.

Legacy Functions

```
void fileRead (string filename, string &contents)
{
    string line;
    ifstream file (filename);

    if (file.is_open())
    {
        while (getline (file, line))
        {
            //cout << line << endl;
            contents = contents + line + "\n";
        }
        file.close();
    }
    else
    {
        cout << "File does not exist" << endl;
    }
}
```

A fileRead() and fileWrite() system was implemented for accessing files in the directory. These functions are used to read input files and parse them into the necessary input when the pipe or input operator is called.

This function is also used to record the output of a process and write them into an output file when needed.

```
void parsing(string cmd, vector <string> &comms,
            string &c1, string &c2, int &mode)
{
    stringstream ss(cmd);
    string word;
    comms.clear();
    while (ss>>word)
    {
        comms.push_back(word);
    }

    bool found = false;

    for (int i = 0; i < comms.size(); i++)
    {
        if (comms[i] == "|" || comms[i] == ">" || comms[i] == "<")
        {
            if (comms[i] == "|")
            {
                mode = 1;
            }
            else if (comms[i] == ">")
            {
                mode = 2;
            }
            else if (comms[i] == "<")
            {
                mode = 3;
            }

            found = true;

            for (int j=0; j < i; j++)
            {
                c1 = c1 + comms[j] + ' ';
            }

            //cout<<c1<<endl;
        }
    }
}
```

The parsing() function breaks down the string command that the user inputs into key parameters that would be used in the different operations later.

This method is done by applying the stringstream parsing system that was done in previous projects in order to isolate the input commands in groups according to their respective parameters that would be needed when executing the command itself.

This is also useful in identifying the mode of the operation whether it's a pipe ('|'), input ('<'), or output ('>').

Core Functions

```
int pipeFunc(string cmd1, string cmd2, int mode)
{
    int fd1[2]; // first pipe
    int fd2[2]; // second pipe

    int pid;
    string temp;

    if (pipe(fd1) < 0)
    {
        cout<<"Pipe Failed"<<endl;
        return 1;
    }
    if (pipe(fd2) < 0)
    {
```

The pipeFunc() function serves as the core of the Rudimentary Shell program where this function determines the process that is to be executed and how it is going to be executed. It takes three parameters that were previously handled by the parsing() function. Once the input line has been successfully parsed, it passes through the pipeFunc() and executes the commands depending on their modes. ('<', '>', or '|')

```
//Parent Process
else if (pid > 0)
{
    close(fd1[0]); // Close reading end of first pipe
    // Write input string and close writing end of first pipe.
    write(fd1[1], temp.c_str(), temp.length()+1);
    close(fd1[1]);

    wait(NULL);
    read(fd2[0], outbuf, temp.length());
    if (mode == 3) fileWrite(cmd2, outbuf);
    close(fd2[0]);
}

//Child process
else
{
    // Read a string using first pipe
    read(fd1[0], inbuf, temp.length());
    inbuf[temp.length()] = '\0';

    fileWrite("log.txt", inbuf);

    if (mode == 1)
    {
        temp = exec(cmd2 + " " + "log.txt");
        cout <<temp<<endl;
        write(fd2[1], temp.c_str(), temp.length()+1);
    }

    else if (mode == 2)
    {
        fileWrite(cmd2, inbuf);
        write(fd2[1], temp.c_str(), temp.length()+1);
    }
}
```

This process also handles the Parent-Child process relationship that assists the shell when performing piped commands. Otherwise, the process proceeds with either the input or output based operators.

Simulation Output Review

Basic Shell Command Execution

```
/Users/JayMB/Documents/GitHub/engg126-project1-cck $ ls
Archive.zip
ENGG 126 - Project 1 Final Report DRAFT - KEMPIS.pdf
ENGG 126 - Project 1 First Progress Report - KEMPIS.pdf
ENGG 126 - Project 1 Second Progress Report - KEMPIS.pdf
ENGG 126 - Project 1 Third Progress Report - KEMPIS.pdf
aaaaaaah.cpp
draft
draft.cpp
draft.dSYM
in.txt
log.txt
out.txt
output.txt
project1
project1.cpp
project1.dSYM
shell-interpreter
shell-interpreter-macos
shell-interpreter-macos.cpp
shell-interpreter-macos.dSYM
shell-interpreter.cpp
shell-interpreter.dSYM
shell-proj.cpp

/Users/JayMB/Documents/GitHub/engg126-project1-cck $
```

Using the “ls” command from the standard list of UNIX commands, the shell interpreter is able to pass the command to its execute function and is able to list down all the files that are currently in the directory.

Input (‘<’) Operator

```
/Users/JayMB/Documents/GitHub/engg126-project1-cck $ cat < in.txt
addz
x
dirf
h
etw
returnrtyr
unistddfg

hf
thet
yert
yrty
unistdb
dirwf
vector
sdf

/Users/JayMB/Documents/GitHub/engg126-project1-cck $
```

During the parsing process, if the input command contains the ‘<’ operator, it puts the shell interpreter into input mode which requests a command and a file input.

In this case, the command cat is used to display the contents of its input which is the in.txt file

Output (‘>’) Operator

```
/Users/JayMB/Documents/GitHub/engg126-project1-cck $ ls > newFile.txt
txt

/Users/JayMB/Documents/GitHub/engg126-project1-cck $
```

The same ls command was performed but with an output operator that takes the supposed result of the command and prints it into a file named newFile.txt

```
newFile.txt
1 Archive.zip
2 ENGG 126 - Project 1 Final Report DRAFT - KEMPIS.pdf
3 ENGG 126 - Project 1 First Progress Report - KEMPIS.pdf
4 ENGG 126 - Project 1 Second Progress Report - KEMPIS.pdf
5 ENGG 126 - Project 1 Third Progress Report - KEMPIS.pdf
6 aaaaaaah.cpp
7 draft
8 draft.cpp
9 draft.dSYM
10 in.txt
11 log.txt
12 newFile.txt
13 out.txt
14 output.txt
15 project1
16 project1.cpp
17 project1.dSYM
18 shell-interpreter
19 shell-interpreter-macos
20 shell-interpreter-macos.cpp
21 shell-interpreter-macos.dSYM
22 shell-interpreter.cpp
23 shell-interpreter.dSYM
24 shell-proj.cpp
25
```

```
in.txt
log.txt
newFile.txt
out.txt
output.txt
project1
project1.cpp
shell-interpreter
shell-interprete...
shell-interprete... M
```

Comparing from the basic input command output from above, the `>` operator requests for a file destination or a filename on where the data should go after it is processed. It then takes all the output of the command and processes it in this newly created file. For existing files, this data would overwrite the previous data in the file.

Pipe (`|`) Operator

```
/Users/JayMB/Documents/GitHub/engg126-project1-cck $ cat in.txt |
sort

addz
dirf
dirwf
etw
h
hf
returnrtyr
sdf
thet
unistdbs
unistddfg
vector
x
yert
yrty

/Users/JayMB/Documents/GitHub/engg126-project1-cck $
```

The cat command is used to display the content of the readable file

The pipe command is implemented by taking the output of the first command on the left side of the operator as the input for the command to the right of the `|` operator.

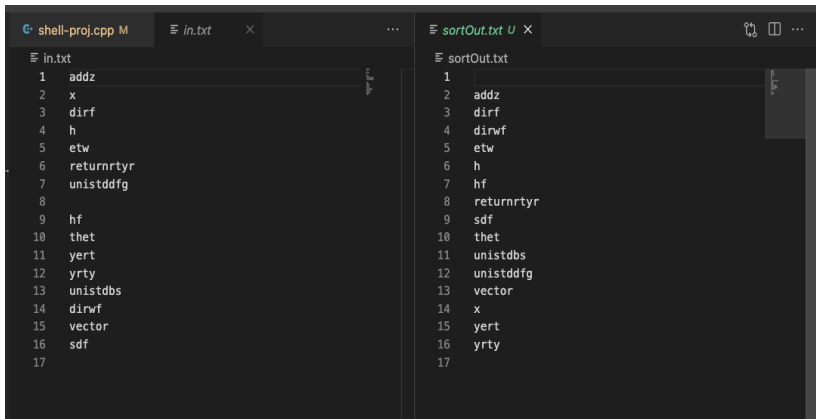
For this case, the cat command reads the in.txt file and takes the output as the input to the sort command which displays the sorted list of words on the terminal

```

/Users/jaym1/Library/CloudStorage/GoogleDrive-joshua.kempism1@gmail.com/My Drive/ENGG 126/engg126-project1-cck
$ cat in.txt | sort -o SortOut.txt
=====

/Users/jaym1/Library/CloudStorage/GoogleDrive-joshua.kempism1@gmail.com/My Drive/ENGG 126/engg126-project1-cck
$

```

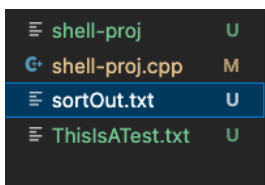


File	Line 1	Line 2	Line 3	Line 4	Line 5	Line 6	Line 7	Line 8	Line 9	Line 10	Line 11	Line 12	Line 13	Line 14	Line 15	Line 16	Line 17
in.txt	addz	x	dirf	h	etw	returnrtyr	unistddfg	hf	thet	yert	yrtty	unistdb	dirwf	vector	sdf		
sortOut.txt	addz	dirf	dirwf	etw	h	hf	returnrtyr	sdf	thet	unistdb	unistddfg	vector	x	yert	yrtty		

Performing the same pipe command with the “ -o “ argument and a filename results in the output of the command to be stored in a new file. This process fulfills the pipe implementation of the rudimentary shell as well as its implementation to accept arguments and parse the commands accordingly.

LEFT: in.txt

RIGHT: (sorted) sortOut.txt



Error Handling

```

/Users/JayMB/Documents/GitHub/engg126-project1-cck $ nnl
sh: nnl: command not found

/Users/JayMB/Documents/GitHub/engg126-project1-cck $

```

Suppose the user enters a command that cannot be recognized by the shell interpreter, it returns an error message and returns back to requesting a new input from the user

Documentation Link

The GitHub repository for the project is made publicly available through the following:

GitHub Link

<https://github.com/jaykempis/engg126-project1-cck>

References

@101010110101, (2009). *Writing my own shell... stuck on pipes?* Stack Overflow. Retrieved February 21, 2023, from

<https://stackoverflow.com/questions/1461331/writing-my-own-shell-stuck-on-pipes>

GeeksforGeeks. (2022). I/O redirection in C++. GeeksforGeeks. Retrieved February 21, 2023, from

<https://www.geeksforgeeks.org/io-redirection-c/>

Gerend, J. (2023). Sort. Microsoft Learn. Retrieved February 23, 2023, from

<https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/sort>

Riehemann, S. (2001). Basic UNIX commands. Computing Information for Stanford Linguists. Retrieved February 23, 2023, from

<https://mally.stanford.edu/~sr/computing/basic-unix.html>