Assignment 1 – Mini Shell

To implement part one of my assignment I used linked lists and I used two methods. Named the parent function and the child function these functions were to represent the respective process more information on the code used is explained below.

Main function

```
std::list<Command> *commandsList = Parser::Parse(input); // grab the c
for (int x = 0; x < commandsList->size(); x++) // cycle through comman
{
    pipe(cur);
    int pid = fork();
    // std::cout << "first " << pid << std::endl;
    if (pid == 0)
    {
        // you are the child
        // std::cout << "Going into child" << std::endl;
        Child(commandsList, x);
        return 1;
    }
    else if (pid > 0){
        // you are the parent
        // std::cout << "Going into Parent" << std::endl;
        Parent(commandsList, x, pid);
    }else if(pid < 0 ){
        std::cout << "Process fork failed. error: " << std::endl;
        exit(1);
    }
}
}
```

This is the main function loop it starts by grabbing all the commands the commands and creating a new fork for the child process. If it is zero child will run otherwise run parent function and if less than zero there was an error. I'd like to make It clear that I did dup the STDIN and STDOUT info the stdfd array

Parent

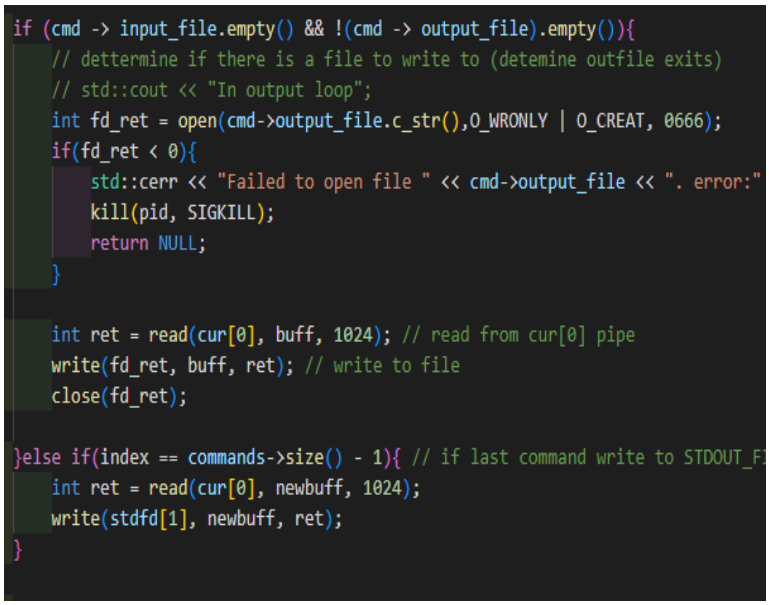
```
int status;
char buff[1024];
char newbuff[1024];

if(index != 0){  // dont need prev in parent
    close(prev[0]);
    close(prev[1]);
}
if(index != commands->size() - 1){ // sets previous pipe
    prev[0] = cur[0];
    prev[1] = cur[1];
}

int Wret = waitpid(pid, &status, 0);
if(Wret == -1 ){
    std::cerr << "Process waitpit() failed. error: " <<
}

// close(cur[1]);
```

This is the start of the parent function, I did some basic things above this photo that I believe will be better if explained in child function. This code snippet is redirects pipes for the pipe command. The buffers will be used to read and write to. And final you can see a waitpid() that waits on a state change from the child, which includes an error checking in case the function fails



```
if (cmd -> input_file.empty() && !(cmd -> output_file).empty()){
    // dettermine if there is a file to write to (detemine outfile exits)
    // std::cout << "In output loop";
    int fd_ret = open(cmd->output_file.c_str(),O_WRONLY | O_CREAT, 0666);
    if(fd_ret < 0){
        std::cerr << "Failed to open file " << cmd->output_file << ". error:"
        kill(pid, SIGKILL);
        return NULL;
    }

    int ret = read(cur[0], buff, 1024); // read from cur[0] pipe
    write(fd_ret, buff, ret); // write to file
    close(fd_ret);

}else if(index == commands->size() - 1){ // if last command write to STDOUT_F
    int ret = read(cur[0], newbuff, 1024);
    write(stdfd[1], newbuff, ret);
}
```

This section of the code is to check for the case that there is a output_file to write to. I open the file, do some error checking in case the file was not correctly opened and finally read from the pipe and write to file.

The if statement below determines if the current command is the last command that way it would write to the terminal and not a file

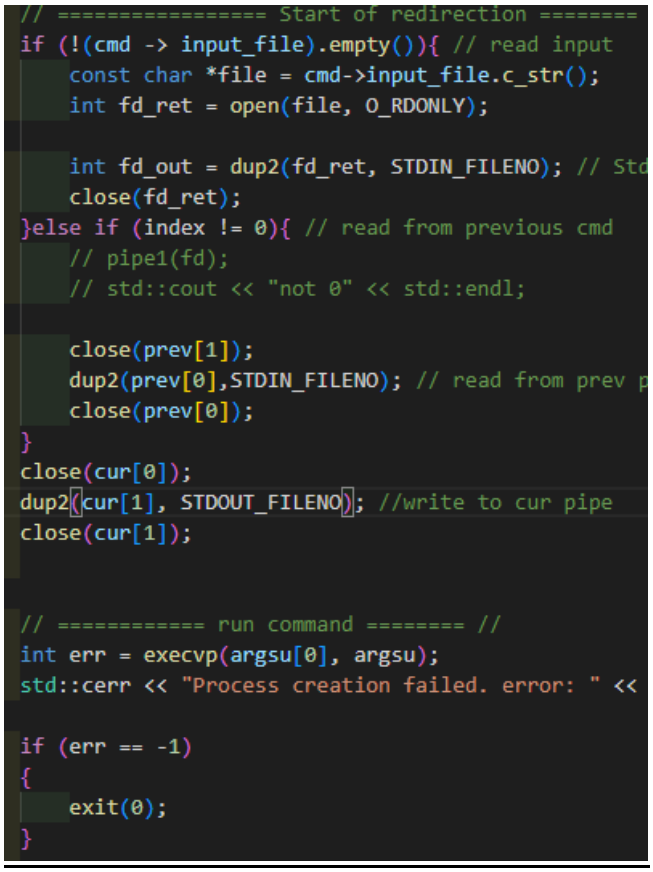
Child

```
auto it = commands->begin(); // start iterating
auto cmd = std::next(it, index);  // grab the current command

// ============ set up to run command ======== //
std::string a = cmd->name; // get the command name
char *C = strdup(a.c_str()); // command name converted to char*

char *argsu[cmd->args.size() + 2]; // create the args
argsu[0] = C;
std::list<std::string>::iterator iter; // iterator for args
int counter = 1;
for (iter = cmd->args.begin(); iter != (cmd->args).end(); iter++){ // loop th
    // load all args into char* array
    argsu[counter] = strdup(iter->c_str());
    counter++; // counter for indexing
}
argsu[sizeof(argsu) / sizeof(argsu[0]) - 1] = NULL; // last index is a NULL
```

The section is the start of the child function , it starts by starting an iterator and then getting the next index which would be the current command. The variable a will represent the command name and it must be converted to a char pointer. Next we create an array that will store all the arguments as well as two extra spaces for the command name as well as the null index. finally we create a for loop to run through all the arguments setting each index in the array to the respective command. finally we set the last index to null.



```
// ================ Start of redirection ========
if (!(cmd -> input_file).empty()){ // read input
    const char *file = cmd->input_file.c_str();
    int fd_ret = open(file, O_RDONLY);

    int fd_out = dup2(fd_ret, STDIN_FILENO); // Std
    close(fd_ret);
}else if (index != 0){ // read from previous cmd
    // pipe1(fd);
    // std::cout << "not 0" << std::endl;

    close(prev[1]);
    dup2(prev[0],STDIN_FILENO); // read from prev p
    close(prev[0]);
}
close(cur[0]);
dup2(cur[1], STDOUT_FILENO); //write to cur pipe
close(cur[1]);


// ============ run command ======== //
int err = execvp(argsu[0], argsu);
std::cerr << "Process creation failed. error: " <<

if (err == -1)
{
    exit(0);
}
```

this is the final section of the child function. the first if statement determines if there is an input file to accept if so open that file and redirect The file description as standard in meaning that the file will now be accepted as standard input then close the file. The other else if statement runs if the index is not the first one. if so we will close the write of the pipe, redirect standard input to pipe 0 and close the pipe. Outside of both if statements we will close the current zero pipe and redirect standard output to be written to current one finally closing current one pipe. once this is done execute the command using execvp. And below that we do some error checking to determine if the command was executed.

Parser.cpp

A big implication of writing code like this is a lack of functions that may cause repetition in writing code and may also make things hard to understand. I have also noticed a large use of auto variable types this may cause a bug that will be difficult to debug as the logic may be correct but type returned from function may be wrong from what is expected. Finally the issue with using a lot standard libraries may deprecation of methods. If a function plays a big role in how your code run and it is deprecated , your code would require a big overhaul and refactor to be able to work with current methods