Custom Shell Implementation

Objective:- Build a simple shell in C++ that can execute commands, manage processes, and handle redirection and piping.

Code:-

```
#include <iostream>
#include <sstream>
#include <vector>
#include <string>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <fcntl.h>
#include <signal.h>
#include <map>
#include <chrono>
#include <pwd.h>
#include <readline/readline.h>
#include <readline/history.h>
#include <termios.h>
#include <cstring>
#include <cstdlib>
#include <numeric>
#include <iterator>
using namespace std;
```

```
struct Job {
    pid_t pid;
    string command;
    bool running;
};
static vector<string> historyList;
static map<int, Job> jobs;
static int jobCounter = 1;
static string lastCommand;
// ------------------------ Helpers ------------------------
void printColored(const string &msg, const char *colorCode = "\033[0m") {
    cout << colorCode << msg << "\033[0m";
}
void printSuccess(const string &msg) { printColored(msg, "\033[1;32m"); cout << '\n'; }
void printError(const string &msg)   { printColored(msg, "\033[1;31m"); cout << '\n'; }
void printInfo(const string &msg)    { printColored(msg, "\033[1;33m"); cout << '\n'; }
// Disable echo and read a password line (portable)
string readPasswordLine() {
    struct termios oldt, newt;
    if (tcgetattr(STDIN_FILENO, &oldt) != 0) {
        // fallback to simple getline if tcgetattr fails
        string pwd;
        getline(cin, pwd);
        return pwd;    }
```

```
    newt = oldt;
    newt.c_lflag &= ~ECHO;
    tcsetattr(STDIN_FILENO, TCSANOW, &newt);
    string pwd;
    getline(cin, pwd);
    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
    return pwd;
}
// Returns current working directory
string getCurrentDir() {
    char buf[1024];
    if (getcwd(buf, sizeof(buf))) return string(buf);
    return string("");
}
// ------------------------- Authentication -------------------------
void authenticateUser() {
    cout << "Enter password to access MyShell: ";
    string pwd = readPasswordLine();
    cout << "\n";
    if (pwd != "admin123") {
        printError("Access Denied: wrong password.");
        exit(1);
    }
    printSuccess("Authentication successful. Welcome to MyShell!");
}
```

```
// ------------------------ Prompt ------------------------
string buildPrompt() {
   struct passwd *pw = getpwuid(getuid());
   string user = pw ? pw->pw_name : "user";
   string cwd = getCurrentDir();
   std::ostringstream ss;
   ss << "\033[1;34m" << user << "\033[0m:\033[1;36m" << cwd << "\033[0m
\033[1;32mMyShell>\033[0m ";
   return ss.str();
}
// ------------------------ Parsing ------------------------
vector<string> parseInput(const string &input) {
   vector<string> tokens;
   istringstream iss(input);
   string tok;
   while (iss >> tok) tokens.push_back(tok);
   return tokens;
}
// ------------------------ I/O Redirection ------------------------
void handleRedirection(vector<string> &args) {
   for (size_t i = 0; i < args.size(); ++i) {
      if (args[i] == ">") {
         if (i + 1 >= args.size()) { printError("Syntax error: > requires a filename");
return; }
         int fd = open(args[i+1].c_str(), O_WRONLY | O_CREAT | O_TRUNC,
0644);
         if (fd < 0) { perror("open"); return; }
```

```
            dup2(fd, STDOUT_FILENO);
            close(fd);
            args.erase(args.begin() + i, args.begin() + i + 2);
            --i;
        } else if (args[i] == "<") {
            if (i + 1 >= args.size()) { printError("Syntax error: < requires a filename");
return; }
            int fd = open(args[i+1].c_str(), O_RDONLY);
            if (fd < 0) { perror("open"); return; }
            dup2(fd, STDIN_FILENO);
            close(fd);
            args.erase(args.begin() + i, args.begin() + i + 2);
            --i;
        }
    }
}
// ------------------------ Pipeline ------------------------
void executePipeline(vector< vector<string> > &commands) {
    int n = commands.size();
    if (n == 0) return;
    vector<int> fds(2*(n-1), 0);
    for (int i = 0; i < n-1; ++i) {
        if (pipe(&fds[i*2]) == -1) { perror("pipe"); return; }
    }
    for (int i = 0; i < n; ++i) {
        pid_t pid = fork();
        if (pid == 0) {
```

```
            // stdin from previous
            if (i != 0) {
                dup2(fds[(i-1)*2], STDIN_FILENO);
            }
            // stdout to next
            if (i != n-1) {
                dup2(fds[i*2 + 1], STDOUT_FILENO);
            }
            // close all fds in child
            for (int j = 0; j < 2*(n-1); ++j) close(fds[j]);
            // handle redirection in this segment
            handleRedirection(commands[i]);
            // prepare argv
            vector<char*> argv;
            for (size_t k = 0; k < commands[i].size(); ++k)
argv.push_back(const_cast<char*>(commands[i][k].c_str()));
            argv.push_back(nullptr);
            execvp(argv[0], argv.data());
            // exec failed
            perror("exec");
            exit(1);
        } else if (pid < 0) {
            perror("fork");
            // close fds and return
            for (int j = 0; j < 2*(n-1); ++j) close(fds[j]);
            return;
        }
```

```
        // parent continues to next command
    }
    // parent: close fds and wait
    for (int j = 0; j < 2*(n-1); ++j) close(fds[j]);
    for (int i = 0; i < n; ++i) wait(NULL);
}
// ------------------------ Job control helpers ------------------------
void printJobs() {
    if (jobs.empty()) { cout << "No background jobs.\n"; return; }
    for (auto &p : jobs) {
        cout << "[" << p.first << "] PID: " << p.second.pid
             << " CMD: " << p.second.command
             << (p.second.running ? " (Running)\n" : " (Stopped)\n");
    }
}
void bringToForeground(int id) {
    auto it = jobs.find(id);
    if (it == jobs.end()) { printError("Job id not found"); return; }
    pid_t pid = it->second.pid;
    int status;
    cout << "Bringing job [" << id << "] PID " << pid << " to foreground\n";
    if (kill(pid, SIGCONT) < 0) perror("kill(SIGCONT)");
    waitpid(pid, &status, 0);
    jobs.erase(it);
}
```

```
void sendToBackground(int id) {
    auto it = jobs.find(id);
    if (it == jobs.end()) { printError("Job id not found"); return; }
    pid_t pid = it->second.pid;
    if (kill(pid, SIGCONT) < 0) { perror("kill"); return; }
    it->second.running = true;
    cout << "Job [" << id << "] resumed in background (PID " << pid << ")\n";
}

// ------------------------ Builtins + External execute ------------------------
void executeCommand(vector<string> args, bool background) {
    if (args.empty()) return;

    // build full command string for history
    string cmdLine;
    for (size_t i = 0; i < args.size(); ++i) {
        if (i) cmdLine += " ";
        cmdLine += args[i];
    }
    historyList.push_back(cmdLine);
    lastCommand = historyList.back();
    // builtins
    string cmd = args[0];
    if (cmd == "cd") {
        if (args.size() > 1) {
            if (chdir(args[1].c_str()) != 0) perror("cd");
```

```
        } else {
            const char *h = getenv("HOME");
            if (h) chdir(h);
        }
        return;
    }
    if (cmd == "exit") {
        printInfo("Exiting MyShell...");
        exit(0);
    }
    if (cmd == "help") {
        cout <<
        "Built-ins: cd, exit, help, history, !!, whoami, setenv, printenv, jobs, fg,
bg\n"
        "Supports: pipes '|', redirection '<' and '>', background '&'.\n";
        return;
    }
    if (cmd == "history") {
        for (size_t i = 0; i < historyList.size(); ++i) cout << i+1 << " " << historyList[i]
<< "\n";
        return;
    }
    if (cmd == "!!") {
        if (lastCommand.empty()) { printError("No previous command"); return; }
        cout << "Repeating: " << lastCommand << "\n";
        vector<string> repeatArgs = parseInput(lastCommand);
        executeCommand(repeatArgs, false);
```

```
        return;
    }
    if (cmd == "whoami") {
        struct passwd *pw = getpwuid(getuid());
        cout << (pw ? pw->pw_name : "unknown") << "\n";
        return;
    }
    if (cmd == "setenv") {
        if (args.size() != 3) { printError("Usage: setenv VAR VALUE"); return; }
        if (setenv(args[1].c_str(), args[2].c_str(), 1) != 0) perror("setenv");
        else printSuccess("Environment variable set");
        return;
    }
    if (cmd == "printenv") {
        if (args.size() == 1) {
            extern char **environ;
            for (char **env = environ; *env; ++env) cout << *env << "\n";
        } else {
            char *v = getenv(args[1].c_str());
            if (v) cout << args[1] << "=" << v << "\n"; else printError("Variable not
set");
        }
        return;
    }
    if (cmd == "jobs") { printJobs(); return; }
    if (cmd == "fg") {
        if (args.size() < 2) { printError("Usage: fg <jobid>"); return; }
```

```
        bringToForeground( stoi(args[1]) );
        return;
    }
    if (cmd == "bg") {
        if (args.size() < 2) { printError("Usage: bg <jobid>"); return; }
        sendToBackground( stoi(args[1]) );
        return;
    }

    // External command
    auto tstart = chrono::high_resolution_clock::now();
    pid_t pid = fork();
    if (pid == 0) {
        // child
        signal(SIGINT, SIG_DFL);
        handleRedirection(args);
        // build argv
        vector<char*> argv;
        for (size_t i = 0; i < args.size(); ++i)
argv.push_back(const_cast<char*>(args[i].c_str()));
        argv.push_back(nullptr);
        execvp(argv[0], argv.data());
        // if execvp returns, it's an error
        perror("execvp");
        exit(127);
    } else if (pid > 0) {
        if (background) {
```

```
            Job j; j.pid = pid; j.command = cmd; j.running = true;
            jobs[jobCounter] = j;
            cout << "[" << jobCounter << "] " << pid << " (background)\n";
            ++jobCounter;
        } else {
            int status = 0;
            waitpid(pid, &status, 0);
            auto tend = chrono::high_resolution_clock::now();
            chrono::duration<double> dur = tend - tstart;
            cout << "\033[1;36mExecution time: " << dur.count() << "s\033[0m\n";
            if (WIFEXITED(status) && WEXITSTATUS(status) == 0) printSuccess("✅ Command completed");
            else printError("❌ Command failed");
        }
    } else {
        perror("fork");
    }
}

// ------------------------ Readline completion (basic) ------------------------
// Use default filename completion.
static char **myshell_completion(const char *text, int start, int end) {
    (void)start; (void)end;
    return rl_completion_matches(text, rl_filename_completion_function);
}
```

```
// ------------------------ Signal handler ------------------------
void sigint_handler(int signo) {
    (void)signo;
    cout << "\n(Use 'exit' to quit)\n";
}

// ------------------------ Main ------------------------
int main() {
    // Authentication
    authenticateUser();

    // Readline setup for history & tab
    rl_bind_key('\t', rl_complete);
    rl_attempted_completion_function = myshell_completion;

    // signal
    signal(SIGINT, sigint_handler);

    while (true) {
        string prompt = buildPrompt();
        char *line_read = readline(prompt.c_str());
        if (!line_read) { cout << "\n"; break; } // EOF (Ctrl+D)
        string line(line_read);
        free(line_read);
   // trim leading/trailing spaces
        size_t first = line.find_first_not_of(" \t\n");
```

```
if (first == string::npos) continue;
size_t last = line.find_last_not_of(" \t\n");
line = line.substr(first, last - first + 1);
if (line.empty()) continue;
add_history(line.c_str());
historyList.push_back(line);
lastCommand = line;

// pipeline?
if (line.find('|') != string::npos) {
   vector< vector<string> > pipeline;
   string segment;
   stringstream ss(line);
   while (getline(ss, segment, '|')) {
      // trim
      size_t f = segment.find_first_not_of(" \t\n");
      if (f == string::npos) continue;
      size_t l = segment.find_last_not_of(" \t\n");
      string seg = segment.substr(f, l-f+1);
      pipeline.push_back(parseInput(seg));
   }
   if (!pipeline.empty()) executePipeline(pipeline);
   continue;
}
```

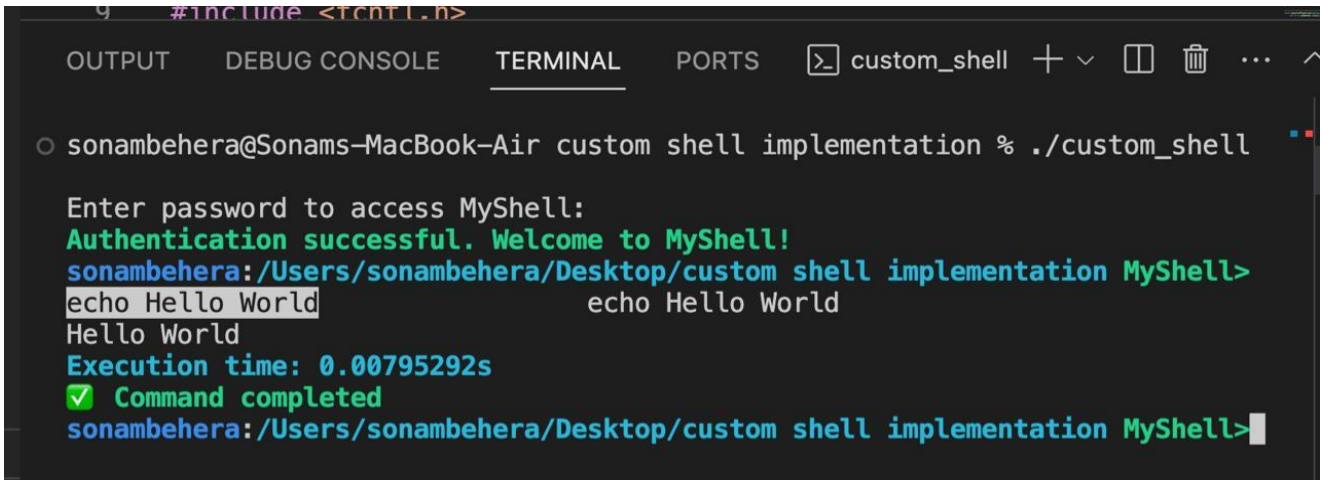
```
  // parse and background check
      vector<string> args = parseInput(line);
      bool background = false;
      if (!args.empty() && args.back() == "&") { background = true;
args.pop_back(); }

      executeCommand(args, background);
   }

   return 0;
}
```

Output :-

- Day1:-input parsing



- Day2:-Basic commands

```
sonambehera:/Users/sonambehera/Desktop/custom shell implementation MyShell>
pwd
/Users/sonambehera/Desktop/custom shell implementation
Execution time: 0.00691808s
sonambehera:/Users/sonambehera/Desktop/custom shell implementation MyShell>
```

```
sonambehera:/Users/sonambehera/Desktop/custom shell implementation MyShell>
pwd
/Users/sonambehera/Desktop/custom shell implementation
sonambehera:/Users/sonambehera/Desktop/custom shell implementation MyShell>
sonambehera:/Users/sonambehera/Desktop MyShell>
```

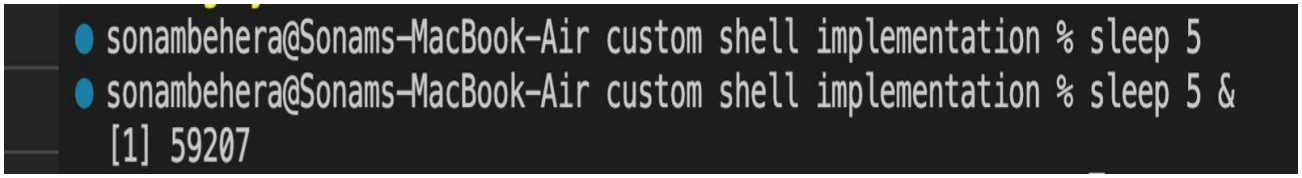
```
sonambehera:/Users/sonambehera/Desktop MyShell> pwd
/Users/sonambehera/Desktop
Execution time: 0.00595163s
sonambehera:/Users/sonambehera/Desktop MyShell>
```

```
sonambehera:/Users/sonambehera/Desktop MyShell> pwd
/Users/sonambehera/Desktop
sonambehera:/Users/sonambehera/Desktop MyShell> ls
File_explorer
HTML TUTORIAL
NODEJS
SQLDeveloper.app
Screenshot 2025-01-25 at 12.02.42 PM.png
Screenshot 2025-07-02 at 6.57.08 PM.png
Screenshot 2025-10-18 at 9.16.54 PM.png
Screenshot 2025-10-24 at 10.52.06 AM.png
Screenshot 2025-10-30 at 10.41.50 PM.png
Screenshot 2025-11-06 at 9.14.22 AM.png
Screenshot 2025-11-08 at 3.07.43 PM.png
```

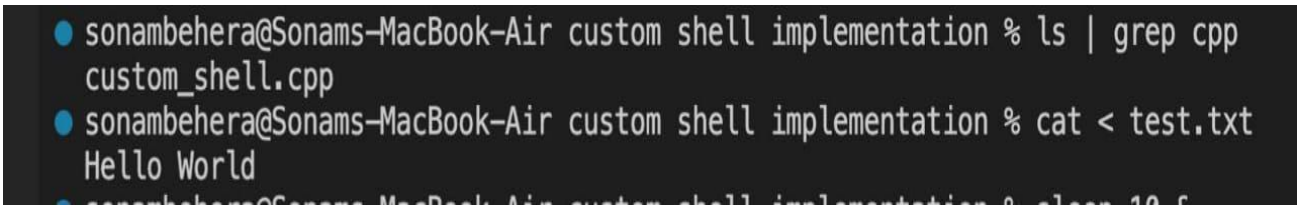
```
women safety
sonambehera:/Users/sonambehera/Desktop MyShell> help
Built-ins: cd, exit, help, history, !!, whoami, setenv, printenv, jobs, fg,
bg
sonambehera:/Users/sonambehera/Desktop MyShell>
```

```
sonambehera:/Users/sonambehera/Desktop MyShell> exit
Exiting MyShell...
sonambehera@Sonams-MacBook-Air custom shell implementation %
```

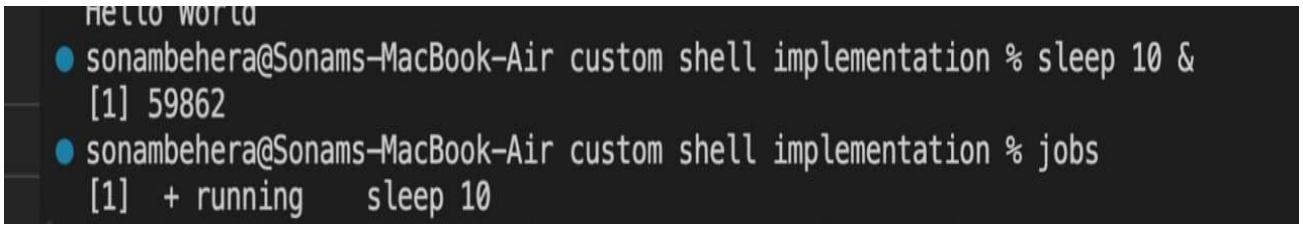
- Day3:-foreground & background jobs



- Day4:-piping and redirection



- Day 5:-job control



Also added advance features like authentication