**Project Code:**

*// command\_line\_arguments.cpp*

#include <iostream>

#include <string.h>

#include <unistd.h>

#include <cstdlib>

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

*// INCLUDING LIBRARIES*

#include <fstream>

#include <vector>

#include <cstdio>

#include <fcntl.h>

#include <cstring>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netinet/ip\_icmp.h>

#include <arpa/inet.h>

#include <netdb.h>

#define TOKENSIZE 100

using namespace std;

void StrTokenizer(char \*line, char \*\*argv);

void myExecvp(char \*\*argv);

int GetEnv();

*// INCLUDING FUNCTION*

void executeCd(char \*\*argv);

void executeCat(char \*\*argv);

void executeCopy(char \*\*argv);

void executeAppend(char \*\*argv);

void executeHistory(const vector<string> &commandHistory);

void executeCmp(char \*\*argv);

void executeRemove(char \*\*argv);

void executeKillCommand(char \*\*argv);

void executeTopCommand(char \*\*argv);

void executeOpenCommand(char \*\*argv);

void executeSwapFiles(char \*\*argv);

void executeLineCount(char \*\*argv);

void executeNetworkDetails();

void executeCalculator(char \*\*argv);

void executeUnitConverter(char \*\*argv);

void executeRandomPasswordGenerator(char \*\*argv);

void executePingCommand(char \*\*argv);

*// MAIN FUNCITON*

int main()

{

    char \*path2;

    char \*arr[250];

    char \*Tokenized;

    char input[250];

    char \*argv[TOKENSIZE];

*// INCLUDING VARIABLE OR OTHER ELEMENTS*

    vector<string> commandHistory; *// CREATING A COMMAND HISTORY VECTOR*

    while (true)

    {

        cout << "cwushell-> ";

        cin.getline(input, 250);

        StrTokenizer(input, argv);

        if (strcmp(input, "exit") == 0)

        {

            break;

        }

        else if (strcmp(input, "") == 0)

        {

            continue;

        }

*// STORING THE EXECUTED COMMAND*

        string command(argv[0]);

        commandHistory.push\_back(command);

        if (strcmp(argv[0], "cd") == 0)

        {

            executeCd(argv);

            continue;

        }

        else if (strcmp(argv[0], "cat") == 0)

        {

            executeCat(argv);

            continue;

        }

        else if (strcmp(argv[0], "copy") == 0)

        {

            executeCopy(argv);

            continue;

        }

        else if (strcmp(argv[0], "append") == 0)

        {

            executeAppend(argv);

            continue;

        }

        else if (strcmp(argv[0], "history") == 0)

        {

            executeHistory(commandHistory);

            continue;

        }

        else if (strcmp(argv[0], "cmp") == 0)

        {

            executeCmp(argv);

            continue;

        }

        else if (strcmp(argv[0], "remove") == 0)

        {

            executeRemove(argv);

            continue;

        }

        else if (strcmp(argv[0], "kill") == 0)

        {

            executeKillCommand(argv);

            continue;

        }

        else if (strcmp(argv[0], "top") == 0)

        {

            executeTopCommand(argv);

            continue;

        }

        else if (strcmp(argv[0], "open") == 0)

        {

            executeOpenCommand(argv);

            continue;

        }

        else if (strcmp(argv[0], "swap") == 0)

        {

            executeSwapFiles(argv);

            continue;

        }

        else if (strcmp(argv[0], "count") == 0)

        {

            executeLineCount(argv);

            continue;

        }

        else if (strcmp(argv[0], "network") == 0)

        {

            executeNetworkDetails();

            continue;

        }

        else if (strcmp(argv[0], "calc") == 0)

        {

            executeCalculator(argv);

            continue;

        }

        else if (strcmp(argv[0], "convert") == 0)

        {

            executeUnitConverter(argv);

            continue;

        }

        else if (strcmp(argv[0], "password") == 0)

        {

            executeRandomPasswordGenerator(argv);

            continue;

        }

        myExecvp(argv);

    }

    return 0;

}

void myExecvp(char \*\*argv)

{

    pid\_t pid;

    int status;

    int childStatus;

    pid = fork();

    if (pid == 0)

    {

        childStatus = execvp(\*argv, argv);

        if (childStatus < 0)

        {

            cout << "ERROR:wrong input" << endl;

        }

        exit(0);

    }

    else if (pid < 0)

    {

        cout << "somthing went wrong!" << endl;

    }

    else

    {

        int status;

        waitpid(pid, &status, 0);

    }

}

void StrTokenizer(char \*input, char \*\*argv)

{

    char \*stringTokenized;

    stringTokenized = strtok(input, " ");

    int i = 0;

    while (stringTokenized != NULL && i < TOKENSIZE - 1)

    {

        argv[i] = stringTokenized;

        stringTokenized = strtok(NULL, " ");

        i++;

    }

    argv[i] = NULL;

}

int GetEnv()

{

    char \*path2;

    char \*arr2[250];

    char \*Tokenized;

    path2 = getenv("PATH");

    Tokenized = strtok(path2, ":");

    int k = 0;

    while (Tokenized != NULL)

    {

        arr2[k] = Tokenized;

        Tokenized = strtok(NULL, ":");

        k++;

    }

    arr2[k] = NULL;

    return \*arr2[k];

}

void executeCd(char \*\*argv)

{

    if (argv[1] == NULL)

    {

*// No argument provided with cd, print an error message*

        cout << "ERROR: No directory specified" << endl;

    }

    else

    {

        cout << "You are inside: " << argv[1] << endl; *// Print the directory name*

        if (chdir(argv[1]) != 0)

        {

*// chdir failed, print an error message*

            cout << "ERROR: Failed to change directory" << endl;

        }

    }

}

void executeCat(char \*\*argv)

{

    if (argv[1] == NULL)

    {

*// No argument provided with cat, print an error message*

        cout << "ERROR: No file specified" << endl;

    }

    else

    {

        ifstream file(argv[1]);

        if (file.is\_open())

        {

            string line;

            while (getline(file, line))

            {

                cout << line << endl;

            }

            file.close();

        }

        else

        {

            cout << "ERROR: Failed to open file" << endl;

        }

    }

}

void executeCopy(char \*\*argv)

{

    if (argv[1] == NULL || argv[2] == NULL)

    {

*// Insufficient arguments provided with copy, print an error message*

        cout << "ERROR: Insufficient arguments provided" << endl;

    }

    else

    {

        ifstream inputFile(argv[1]);

        ofstream outputFile(argv[2]);

        if (inputFile.is\_open() && outputFile.is\_open())

        {

            string line;

            while (getline(inputFile, line))

            {

                outputFile << line << endl;

            }

            cout << "File copied successfully" << endl;

            inputFile.close();

            outputFile.close();

        }

        else

        {

            cout << "ERROR: Failed to open files" << endl;

        }

    }

}

void executeAppend(char \*\*argv)

{

    if (argv[1] == nullptr || argv[2] == nullptr)

    {

*// Insufficient arguments provided with append, print an error message*

        cout << "ERROR: Insufficient arguments provided" << endl;

    }

    else

    {

        string fileName = argv[1];

        ofstream outputFile(fileName, ios::app); *// Open the file in append mode*

        if (outputFile.is\_open())

        {

            for (int i = 2; argv[i] != nullptr; i++)

            {

                outputFile << argv[i] << " ";

            }

            cout << "Text appended successfully" << endl;

            outputFile.close();

        }

        else

        {

            cout << "ERROR: Failed to open file" << endl;

        }

    }

}

void executeHistory(const vector<string> &commandHistory)

{

    cout << "Command History:" << endl;

    for (int i = 0; i < commandHistory.size(); i++)

    {

        cout << i + 1 << ": " << commandHistory[i] << endl;

    }

}

void executeCmp(char \*\*argv)

{

    if (argv[1] == nullptr || argv[2] == nullptr)

    {

        std::cout << "ERROR: Insufficient arguments. Usage: cmp <file1> <file2>" << std::endl;

        return;

    }

    std::ifstream file1(argv[1]);

    std::ifstream file2(argv[2]);

    if (!file1.is\_open() || !file2.is\_open())

    {

        std::cout << "ERROR: Unable to open one or both files." << std::endl;

        return;

    }

    std::string line1, line2;

    int lineNum = 1;

    bool isDifferent = false;

    while (std::getline(file1, line1) && std::getline(file2, line2))

    {

        if (line1 != line2)

        {

            isDifferent = true;

            std::cout << "Files differ: line " << lineNum << std::endl;

            std::cout << "< " << line1 << std::endl;

            std::cout << "> " << line2 << std::endl;

        }

        lineNum++;

    }

    if (std::getline(file1, line1))

    {

        isDifferent = true;

        std::cout << "Files differ: line " << lineNum << std::endl;

        std::cout << "< " << line1 << std::endl;

    }

    else if (std::getline(file2, line2))

    {

        isDifferent = true;

        std::cout << "Files differ: line " << lineNum << std::endl;

        std::cout << "> " << line2 << std::endl;

    }

    file1.close();

    file2.close();

    if (!isDifferent)

    {

        std::cout << "Files are identical." << std::endl;

    }

}

void executeRemove(char \*\*argv)

{

    if (argv[1] == nullptr)

    {

        std::cout << "ERROR: No file specified" << std::endl;

        return;

    }

    std::string filename(argv[1]);

*// Remove the file using the remove() function*

    int result = std::remove(filename.c\_str());

    if (result != 0)

    {

        std::cout << "ERROR: Failed to remove the file" << std::endl;

    }

    else

    {

        std::cout << "File successfully removed: " << filename << std::endl;

    }

}

*// Sam-Work*

void executeCalculator(char \*\*argv)

{

    if (argv[1] == nullptr)

    {

        std::cout << "ERROR: No expression specified" << std::endl;

        return;

    }

    std::string expression(argv[1]);

*// Evaluate the expression using the system function*

    std::string command = "echo \"" + expression + "\" | bc";

    std::string result = "";

    FILE \*pipe = popen(command.c\_str(), "r");

    if (pipe)

    {

        char buffer[128];

        while (!feof(pipe))

        {

            if (fgets(buffer, sizeof(buffer), pipe) != nullptr)

                result += buffer;

        }

        pclose(pipe);

    }

*// Check if there was an error in the calculation*

    if (result.empty())

    {

        std::cout << "ERROR: Failed to evaluate the expression" << std::endl;

    }

    else

    {

        std::cout << "Result: " << result << std::endl;

    }

}

void executeUnitConverter(char \*\*argv)

{

    if (argv[1] == nullptr || argv[2] == nullptr)

    {

        std::cout << "ERROR: Insufficient arguments. Usage: unitconverter <value> <unit>" << std::endl;

        return;

    }

    double value = std::atof(argv[1]);

    std::string unit(argv[2]);

    double result1;

    double result2;

*// Convert units based on the specified unit*

    if (unit == "km")

    {

        result1 = value \* 1000;

        result2 = value \* 0.621371;

        std::cout << value << " km = " << result1 << " m" << std::endl;

        std::cout << value << " km = " << result2 << " mi" << std::endl;

    }

    else if (unit == "m")

    {

        result1 = value / 1000;

        result2 = value \* 0.000621371;

        std::cout << value << " m = " << result1 << " km" << std::endl;

        std::cout << value << " m = " << result2 << " mi" << std::endl;

    }

    else if (unit == "mi")

    {

        result1 = value \* 1.60934;

        result2 = value \* 1609.34;

        std::cout << value << " mi = " << result1 << " km" << std::endl;

        std::cout << value << " mi = " << result2 << " m" << std::endl;

    }

    else if (unit == "km")

    {

        result1 = value / 1.60934;

        result2 = value \* 1000;

        std::cout << value << " km = " << result1 << " mi" << std::endl;

        std::cout << value << " km = " << result2 << " m" << std::endl;

    }

    else if (unit == "kg")

    {

        result1 = value \* 1000;

        std::cout << value << " kg = " << result1 << " g" << std::endl;

    }

    else if (unit == "g")

    {

        result1 = value / 1000;

        std::cout << value << " g = " << result1 << " kg" << std::endl;

    }

    else

    {

        std::cout << "ERROR: Unsupported unit. Available units: km, m, mi, kg, g" << std::endl;

    }

}

std::string generateRandomPassword(int length)

{

    std::string password;

    const std::string characters = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!@#$%^&\*()";

    int charCount = characters.length();

    srand(static\_cast<unsigned int>(time(nullptr)));

    for (int i = 0; i < length; ++i)

    {

        password += characters[rand() % charCount];

    }

    return password;

}

void executeRandomPasswordGenerator(char \*\*argv)

{

    if (argv[1] == nullptr)

    {

        std::cout << "ERROR: No password length specified" << std::endl;

        return;

    }

    int length = std::stoi(argv[1]);

    if (length <= 0)

    {

        std::cout << "ERROR: Invalid password length" << std::endl;

        return;

    }

    std::string password = generateRandomPassword(length);

    std::cout << "Randomly generated password: " << password << std::endl;

}

*// USING SYSTEM CALL*

void executeKillCommand(char \*\*argv)

{

    if (argv[1] == NULL)

    {

        std::cout << "ERROR: No process name specified" << std::endl;

    }

    else

    {

        std::string processName(argv[1]);

        std::string command = "pkill " + processName;

        int result = system(command.c\_str());

        if (result == 0)

        {

            std::cout << "Processes terminated successfully." << std::endl;

        }

        else

        {

            std::cout << "Failed to terminate processes." << std::endl;

        }

    }

}

void executeTopCommand(char \*\*argv)

{

    if (argv[1] == nullptr)

    {

        cout << "ERROR: Process name not provided" << endl;

        return;

    }

    string processName = argv[1];

    string command = "top -n 1 -p $(pgrep \"" + processName + "\")";

    system(command.c\_str());

}

void executeOpenCommand(char \*\*argv)

{

    if (argv[1] == nullptr)

    {

        cout << "ERROR: Website name not provided" << endl;

        return;

    }

    string websiteName = argv[1];

*// Format the website name as a URL search query*

    string url = "https://www.google.com/search?q=" + websiteName;

*// Open the URL in Firefox*

    string command = "firefox -new-tab " + url;

    system(command.c\_str());

}

void executeSwapFiles(char \*\*argv)

{

    const char \*file1 = argv[1];

    const char \*file2 = argv[2];

    int pipefd[2];

    if (pipe(pipefd) == -1)

    {

        std::cout << "ERROR: Pipe creation failed" << std::endl;

        return;

    }

    pid\_t childPid = fork();

    if (childPid < 0)

    {

        std::cout << "ERROR: Fork failed" << std::endl;

        return;

    }

    else if (childPid == 0)

    {

*// Child process - swap the file contents*

*// Read from file1*

        std::ifstream inputFile1(file1);

        if (!inputFile1.is\_open())

        {

            std::cout << "ERROR: Failed to open file: " << file1 << std::endl;

            return;

        }

        std::string content1((std::istreambuf\_iterator<char>(inputFile1)),

                             (std::istreambuf\_iterator<char>()));

        inputFile1.close();

*// Read from file2*

        std::ifstream inputFile2(file2);

        if (!inputFile2.is\_open())

        {

            std::cout << "ERROR: Failed to open file: " << file2 << std::endl;

            return;

        }

        std::string content2((std::istreambuf\_iterator<char>(inputFile2)),

                             (std::istreambuf\_iterator<char>()));

        inputFile2.close();

*// Write to file1*

        std::ofstream outputFile1(file1);

        if (!outputFile1.is\_open())

        {

            std::cout << "ERROR: Failed to open file: " << file1 << std::endl;

            return;

        }

        outputFile1 << content2;

        outputFile1.close();

*// Write to file2*

        std::ofstream outputFile2(file2);

        if (!outputFile2.is\_open())

        {

            std::cout << "ERROR: Failed to open file: " << file2 << std::endl;

            return;

        }

        outputFile2 << content1;

        outputFile2.close();

        exit(EXIT\_SUCCESS);

    }

    else

    {

*// Parent process*

        int status;

        waitpid(childPid, &status, 0); *// Wait for the child process to finish*

        std::cout << "Swapped the contents of " << file1 << " and " << file2 << std::endl;

    }

}

void executeLineCount(char \*\*argv)

{

    if (argv[1] == nullptr)

    {

        std::cout << "ERROR: Insufficient argument. Usage: linecount <filename>" << std::endl;

        return;

    }

    int pipefd[2];

    if (pipe(pipefd) == -1)

    {

        std::cout << "ERROR: Pipe creation failed" << std::endl;

        return;

    }

    pid\_t childPid = fork();

    if (childPid < 0)

    {

        std::cout << "ERROR: Fork failed" << std::endl;

        return;

    }

    else if (childPid == 0)

    {

*// Child process*

        close(pipefd[0]); *// Close the read end of the pipe*

*// Redirect the output of the file to the write end of the pipe*

        dup2(pipefd[1], STDOUT\_FILENO);

        close(pipefd[1]); *// Close the write end of the pipe*

        execlp("cat", "cat", argv[1], NULL); *// Execute the cat command to read the file*

        exit(EXIT\_FAILURE);

    }

    else

    {

*// Parent process*

        close(pipefd[1]); *// Close the write end of the pipe*

        int lineCount = 0;

        char buffer[4096];

        ssize\_t bytesRead;

*// Read the output of the cat command from the read end of the pipe*

        while ((bytesRead = read(pipefd[0], buffer, sizeof(buffer))) > 0)

        {

            for (ssize\_t i = 0; i < bytesRead; ++i)

            {

                if (buffer[i] == '\n')

                {

                    ++lineCount;

                }

            }

        }

        close(pipefd[0]); *// Close the read end of the pipe*

        int status;

        waitpid(childPid, &status, 0); *// Wait for the child process to finish*

        std::cout << "Number of lines in " << argv[1] << ": " << lineCount << std::endl;

    }

}

void executeNetworkDetails()

{

    FILE \*pipe = popen("ifconfig", "r");

    if (!pipe)

    {

        std::cout << "ERROR: Failed to execute ifconfig command" << std::endl;

        return;

    }

    char buffer[128];

    std::string result;

    while (fgets(buffer, sizeof(buffer), pipe) != nullptr)

    {

        result += buffer;

    }

    pclose(pipe);

    std::cout << result << std::endl;

}

void executePingCommand(char \*\*argv)

{

    if (argv[1] == nullptr)

    {

        cout << "ERROR: Host name/IP address not provided" << endl;

        return;

    }

    string host = argv[1];

*// Resolve the hostname to an IP address*

    struct addrinfo hints, \*res;

    memset(&hints, 0, sizeof(hints));

    hints.ai\_family = AF\_INET;

    int status = getaddrinfo(host.c\_str(), nullptr, &hints, &res);

    if (status != 0)

    {

        cout << "ERROR: Failed to resolve host" << endl;

        return;

    }

    struct sockaddr\_in \*host\_addr = (struct sockaddr\_in \*)res->ai\_addr;

    char \*ip = inet\_ntoa(host\_addr->sin\_addr);

*// Create a raw socket*

    int sockfd = socket(AF\_INET, SOCK\_RAW, IPPROTO\_ICMP);

    if (sockfd < 0)

    {

        cout << "ERROR: Failed to create socket" << endl;

        return;

    }

*// Prepare the ICMP echo request packet*

    const int packet\_size = sizeof(struct icmphdr);

    char packet[packet\_size];

    memset(packet, 0, packet\_size);

    struct icmphdr \*icmp\_header = (struct icmphdr \*)packet;

    icmp\_header->type = ICMP\_ECHO;

    icmp\_header->code = 0;

    icmp\_header->checksum = 0;

    icmp\_header->un.echo.id = getpid();

    icmp\_header->un.echo.sequence = 1;

    icmp\_header->checksum = htons(0xFFFF - (ICMP\_ECHO << 8));

*// Send the ICMP echo request packet*

    if (sendto(sockfd, packet, packet\_size, 0, (struct sockaddr \*)host\_addr, sizeof(struct sockaddr)) <= 0)

    {

        cout << "ERROR: Failed to send ICMP echo request" << endl;

        close(sockfd);

        return;

    }

*// Receive and process ICMP echo reply packets*

    const int recv\_buffer\_size = 1024;

    char recv\_buffer[recv\_buffer\_size];

    struct sockaddr\_in sender\_addr;

    socklen\_t sender\_addr\_len = sizeof(sender\_addr);

    bool received\_reply = false;

    for (int i = 0; i < 5; i++)

    {

*// Wait for ICMP echo reply packet*

        if (recvfrom(sockfd, recv\_buffer, recv\_buffer\_size, 0, (struct sockaddr \*)&sender\_addr, &sender\_addr\_len) > 0)

        {

*// Check if the received packet is an ICMP echo reply*

            struct icmphdr \*icmp\_reply = (struct icmphdr \*)recv\_buffer;

            if (icmp\_reply->type == ICMP\_ECHOREPLY && icmp\_reply->un.echo.id == getpid())

            {

                received\_reply = true;

                cout << "Received ICMP echo reply from " << ip << endl;

                break;

            }

        }

    }

    if (!received\_reply)

    {

        cout << "No ICMP echo reply received from " << ip << endl;

    }

*// Clean up*

    close(sockfd);

    freeaddrinfo(res);

}