**DATASTRUCTURES LAB PROJECT**

**Code:**

#include <iostream>

#include <string>

#include <vector>

using namespace std;

class Repository;

class User;

class Commit {

private:

string message;

string author;

string timestamp;

public:

Commit(string \_message, string \_author, string \_timestamp)

: message(\_message), author(\_author), timestamp(\_timestamp) {}

string getMessage() const {

return message;

}

string getAuthor() const {

return author;

}

string getTimestamp() const {

return timestamp;

}

};

class File {

private:

string name;

string content;

public:

File(string \_name, string \_content) : name(\_name), content(\_content) {}

string getName() const {

return name;

}

string getContent() const {

return content;

}

};

class FileNode {

private:

File\* file;

vector<FileNode\*> children;

public:

FileNode(File\* \_file) : file(\_file) {}

File\* getFile() const {

return file;

}

const vector<FileNode\*>& getChildren() const {

return children;

}

void addChild(FileNode\* child) {

children.push\_back(child);

}

};

class Repository {

private:

string name;

bool isPublic;

User\* owner;

vector<Commit\*> commits;

FileNode\* root; // Root of the file tree

public:

Repository(string \_name, bool \_isPublic, User\* \_owner)

: name(\_name), isPublic(\_isPublic), owner(\_owner), root(nullptr) {}

string getName() const {

return name;

}

bool getVisibility() const {

return isPublic;

}

User\* getOwner() const {

return owner;

}

void addCommit(Commit\* commit) {

commits.push\_back(commit);

}

const vector<Commit\*>& getCommits() const {

return commits;

}

void addFile(File\* file, const string& path) {

if (!root) {

root = new FileNode(nullptr); // Root node is empty

}

FileNode\* currentNode = root;

vector<string> directories = splitPath(path);

for (const auto& dir : directories) {

FileNode\* childNode = findChildNode(currentNode, dir);

if (!childNode) {

File\* placeholder = new File(dir, ""); //

childNode = new FileNode(placeholder);

currentNode->addChild(childNode);

}

currentNode = childNode;

}

currentNode->addChild(new FileNode(file));

}

File\* getFile(const string& path) {

FileNode\* node = getNodeByPath(path);

if (node && node->getFile()) {

return node->getFile();

}

return nullptr;

}

private:

vector<string> splitPath(const string& path) {

vector<string> directories;

string directory;

for (char c : path) {

if (c == '/') {

if (!directory.empty()) {

directories.push\_back(directory);

directory.clear();

}

} else {

directory += c;

}

}

if (!directory.empty()) {

directories.push\_back(directory);

}

return directories;

}

FileNode\* findChildNode(FileNode\* parentNode, const string& name) {

for (FileNode\* child : parentNode->getChildren()) {

if (child->getFile()->getName() == name) {

return child;

}

}

return nullptr;

}

FileNode\* getNodeByPath(const string& path) {

vector<string> directories = splitPath(path);

FileNode\* currentNode = root;

for (const auto& dir : directories) {

currentNode = findChildNode(currentNode, dir);

if (!currentNode) {

return nullptr;

}

}

return currentNode;

}

};

class User {

private:

string username;

string password;

vector<Repository\*> repositories;

vector<User\*> followers;

public:

User(string \_username, string \_password) : username(\_username), password(\_password) {}

string getUsername() const {

return username;

}

string getPassword() const {

return password;

}

void addFollower(User\* follower) {

followers.push\_back(follower);

}

void removeFollower(User\* follower) {

for (auto it = followers.begin(); it != followers.end(); ++it) {

if (\*it == follower) {

followers.erase(it);

break;

}

}

}

const vector<User\*>& getFollowers() const {

return followers;

}

void createRepository(string repoName, bool isPublic) {

repositories.push\_back(new Repository(repoName, isPublic, this));

}

void deleteRepository(string repoName) {

for (auto it = repositories.begin(); it != repositories.end(); ++it) {

if ((\*it)->getName() == repoName) {

delete \*it;

repositories.erase(it);

break;

}

}

}

const vector<Repository\*>& getRepositories() const {

return repositories;

}

};

void createUser(vector<User\*>& users) {

string username, password;

cout << "Enter username: ";

cin >> username;

cout << "Enter password: ";

cin >> password;

users.push\_back(new User(username, password));

cout << "User created: " << username << endl;

}

User\* loginUser(const vector<User\*>& users) {

string username, password;

cout << "Enter username: ";

cin >> username;

cout << "Enter password: ";

cin >> password;

for (auto& user : users) {

if (user->getUsername() == username && user->getPassword() == password) {

cout << "Logged in as: " << user->getUsername() << endl;

return user;

}

}

cout << "Invalid username or password." << endl;

return nullptr;

}

void createRepository(User\* user) {

if (!user) {

cout << "Please log in first." << endl;

return;

}

string repoName;

char visibilityChoice;

bool isPublic;

cout << "Enter repository name: ";

cin >> repoName;

cout << "Is the repository public? (yes/no): ";

cin >> visibilityChoice;

isPublic = (visibilityChoice == 'y' || visibilityChoice == 'Y');

user->createRepository(repoName, isPublic);

cout << "Repository created: " << repoName << endl;

}

void addCommit(User\* user) {

if (!user) {

cout << "Please log in first." << endl;

return;

}

string repoName, message, timestamp;

cout << "Enter repository name: ";

cin >> repoName;

Repository\* repo = nullptr;

for (auto& r : user->getRepositories()) {

if (r->getName() == repoName) {

repo = r;

break;

}

}

if (!repo) {

cout << "Repository not found." << endl;

return;

}

cout << "Enter commit message: ";

cin.ignore();

getline(cin, message);

cout << "Enter commit timestamp: ";

getline(cin, timestamp);

Commit\* commit = new Commit(message, user->getUsername(), timestamp);

repo->addCommit(commit);

cout << "Commit added." << endl;

}

void addFile(User\* user) {

if (!user) {

cout << "Please log in first." << endl;

return;

}

string repoName, fileName, filePath, content;

cout << "Enter repository name: ";

cin >> repoName;

Repository\* repo = nullptr;

for (auto& r : user->getRepositories()) {

if (r->getName() == repoName) {

repo = r;

break;

}

}

if (!repo) {

cout << "Repository not found." << endl;

return;

}

cout << "Enter file name: ";

cin >> fileName;

cout << "Enter file path: ";

cin >> filePath;

cout << "Enter file content: ";

cin.ignore();

getline(cin, content);

File\* file = new File(fileName, content);

repo->addFile(file, filePath);

cout << "File added." << endl;

}

void viewRepositoryFiles(User\* user) {

if (!user) {

cout << "Please log in first." << endl;

return;

}

string repoName, filePath;

cout << "Enter repository name: ";

cin >> repoName;

Repository\* repo = nullptr;

for (auto& r : user->getRepositories()) {

if (r->getName() == repoName) {

repo = r;

break;

}

}

if (!repo) {

cout << "Repository not found." << endl;

return;

}

cout << "Enter file path: ";

cin >> filePath;

File\* file = repo->getFile(filePath);

if (file) {

cout << "File content: " << file->getContent() << endl;

} else {

cout << "File not found." << endl;

}

}

int main() {

vector<User\*> users;

User\* loggedInUser = nullptr;

char choice;

do {

cout << "Menu:\n";

cout << "1. Create User\n";

cout << "2. Log In\n";

cout << "3. Create Repository\n";

cout << "4. Add Commit\n";

cout << "5. Add File\n";

cout << "6. View Repository Files\n";

cout << "7. Log Out\n";

cout << "8. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case '1':

createUser(users);

break;

case '2':

loggedInUser = loginUser(users);

break;

case '3':

createRepository(loggedInUser);

break;

case '4':

addCommit(loggedInUser);

break;

case '5':

addFile(loggedInUser);

break;

case '6':

viewRepositoryFiles(loggedInUser);

break;

case '7':

loggedInUser = nullptr;

cout << "Logged out." << endl;

break;

case '8':

cout << "Exiting..." << endl;

break;

default:

cout << "Invalid choice. Please try again." << endl;

}

} while (choice != '8');

// Cleaning up memory

for (auto& user : users) {

delete user;

}

return 0;

}