VCS.docx

by Aaenoor Saqib Chaudury

GitHub link:https://github.com/Aaenoor/VCS-RE

Submission date: 18-Apr-2024 11:13PM (UTC+0300)
Submission ID: 2352689290
File name: VCS.docx (330.34K)
Word count: 2709
Character count: 16859

VCS

Table of Contents

Introduction 4
Overview of Version Control Systems (VCS) 4
Characteristics of the Version Control System (VCS) 6
Justification of Hashing Design in the Version Control System 8
Incorporating VCS with Metadata Management System 10
Screenshots and descriptions of your code 12
Summary 18
References 19

Table of Figures

Figure 1: Definition of Commit Structure 12
Figure 2: Initialization and Committing Changes of VCS 13
Figure 3: Going back to earlier Version of a File and Hash Calculation 14
Figure 4: Timestamp, Reading, and Writing File Content Utility Functions 15
Figure 5: Function to read, write and calculate 16
Figure 6: Main Function 17

Introduction

Version control systems (VCS) recreate an integral function in the involved landscape of software maturation, are indispensable for expected and analytic project administration. Integrated mechanisms that scrutinize modifications, enrich conspiracy, and maintain project history are compulsory as technology develops. This analysis is founded on the design and management of the fundamental and efficient vector clip system of C++. The project includes essence functionality that delivers avant-garde characteristics such as scrutinizing files, perpetrating shifts, inspecting the commit log, cruising back versions, initializing depositories, and substantiating hashing range. This project analyzes the elaborate repairpersons of formula control, a key factor of current software engineering, as well as demonstrating expertness in C++ programming. (AbediniAla, 2022).

Overview of Version Control Systems (VCS)

Software expansion squads utilize what are anointed version control systems (VCS), or occasionally source code surveillance or modify control systems, to handle software code modifications over a span. VCS is naturally used to chase transformations, enrich squad arrangement, and possess a narrative of project narrative (Janardhanan, 2020).

Key Concepts:

1. **Versioning:**
    - VCS permits architects to assemble versions or photographs of their codebase at distinguishable junctures in time. Each iteration illustrates an individual cycle, conveying a distinct stage of an undertaking.
2. **Collaboration:**
    - VCS permits numerous inventors to cooperate on a scheme at the exact time. This enables determining confrontations when considerable team associates make transformations to the identical file.

3. **History Tracking:**

    - The log, which registers every transformation made to the codebase, furnishes a comprehensive history of all adoptions. Originators can reexamine iterations of more premature times.

4. **Branching and Merging:**

    - VCS furnishes branching, which permits inventors to designate similar outcome rivulets. Branches can be effortlessly melded back into the leading conception to hurry up the evolution of characteristics and bug payoffs. (Chin, 2022).

5. **Concurrency Control:**

    - By handling numerous updates of the exact file at the exact time with VCS, confrontations are detoured and confirmed in a effortless mutual procedure discharge.

Kinds of VCS:

1. **Center VCS (CVCS):**

    - Sustains repository on a prominent server. After reworking files, Pioneers examine the shifts and then incorporate them rear into the preparatory waitperson.

2. **Distributed Version Control Systems (DVCS):**

    - Each innovator has a regional copy of the whole repository. Local commits of modifications can thereupon be synchronized with the foremost server. A widespread disseminated vision control system is anointed Git.

Benefits of VCS:

1. **Collaborative Development:**

    - Allows different developers to collaborate on the same project without getting in each other's way.

2. **Traceability:**

   - Maintains a complete log of changes, which ensures accountability and traceability.

3. **Reproducibility:**

   - Allows engineers to reproduce code versions, which simplifies testing and debugging (Aniche, 2021).

4. **Risk Mitigation:**

   - Minimizes the risk of data loss through the maintenance of a comprehensive record of alterations over time.

5. **Experimentation:**

   - Using branches gives developers the flexibility to analyze all the new characteristics without making any modification in the main code base.

Characteristics of the Version Control System (VCS)

1. Assemble a untouched repository:

- Authorizes diverse innovators that cooperate on the same undertaking without conveying in separate other's way.

2. Back Review:

- Sustains a complete log of transformations, which confirms accountability and traceability.

3. Commit Modification:

- Allow the users to make changes in these files:
  - Total time for the commit.
  - Detail of all changes in the message of commit.
  - All the data of transformed file.

- Commit hash for content warranty.

4. Display Commit Log:

- Provide a detailed log to user that contain summary of all the verified changes. This possesses less influential fragments such as the commit directive, hashes, timestamps, and adjusted files.

5. Go back to previous versions:

- Allow users to retrogress files to exhaustive commits, habilitating their content to versions. (Schultz, 2019).

6. Hash Validation:

- Utilizing hash technique to ascertain content assiduousness between metamorphoses:
    - Generate a diverse hash for a piece commit, founded on the range of all commit.

7. Confirming Integrity:

- Accomplish integrity appraisals to validate the pursuing:
    - Conservation of data in accumulated commits.
    - Scarcity of exact vestibules within the interpretation control archives.

8. Management of Matadata:

- Store metadata associated with each commit, including:
    - Time and date of commit.
    - Detail of commit.
    - Name of files that are affected by commit.
    - Data of all the altered file.
    - Hash validation of commit.

9. GUI (Optional):

- Contemplate assembling a fundamental interface of the person to construct it simply for the person to function with the VCS.

10. Documentation:

- Deliver entire user and innovator documentation on version control system use approaches and components.

11. Fault Finding:

- Bill vigorous blunder detection to abide by unanticipated safeness circumstances and furnish understandable retrievals to users.

12. Validation:

- Designate a rugged validation framework to ascertain the exactitude and dependability of the VCS (Xu, 2020).

Justification of Hashing Design in the Version Control System

Hashing recreates a consequential function in the conviction and guard of version control systems (VCS) for trustworthiness and protection. The determination to employ hashing hinges on specific key facets, such as:

1. Validation of Content Integrity:

- **Goal:** File ranges are not permitted to reverse between commits.
- **Rationale:** Hashing assembles exceptional identifiers or buzzes for each commit, which stimulates content comparison. Any transformations to the file's contents render a new hash importance, permitting an entire tribunal appraisal of the data (Elliott, 2022).

2. Maintaining Data Integrity Across Commits:

- **Goal:** Preserve viscosity in data continuity and dissuade depravity in version control history.

- **Rationale:** Hashing assembles a cryptographic chain converging separately commit, confirming the safeness of respective commits and the entire commit narrative. This defense shields against indecency against data continuity and assures the dependability of version chronology.

3. Efficient Detection of Duplicates:

- **Goal:** Determinate identical narratives and eradicate them fast.
- **Rationale:** Hashing regales each commit as exceptional, furnishing instantaneous designation of reprised records. Corresponding hash significances designate the identical content, startle data administration, and guarantee the security of concise version chronology (Lytvyn, 2019).

4. Detection of Tampering Attempts:

- **Goal**: Identification of unauthorized system modifications or problems.
- **Rationale:** Hashing operates as a tamper-proof agency; Any transformation in the commit content will provoke the hash worth to switch, signaling imaginable security infringements. This constructs the system vigorously against adversarial procedures and upholds version history safeness.

5. Ensuring Consistent Verification:

- **Goal:** Providing efficient and quality content confirmation.
- **Rationale:** Vigorous hashing algorithms, such as SHA-256, provide invariant outcomes across diverse commits and surroundings. This ingrains conviction that no significance of the scale or complicatedness of the version control system, it homogenizes content confirmation.

6. Optimization for Efficiency and Performance:

- **Goal:** Enabling speedy and efficient content confirmation.
- **Rationale:** Hashing algorithms are configured for speedy hash estimations, which decline computational nuances during protection appraisals. This interpretation is demanded, primarily

as the version control chronology extends in volume, and furnishes a convenient and efficacious data guarantee.

Incorporating VCS with Metadata Management System

Melding a version control system (VCS) with a metadata depository is vital to robustly conserve the chronological category of nonfictional content and enhance team conspiracy. This function depicts how metadata is structured and organized within the procedure:

1. Commit Structure:

- Individually commit is associated with a structured metadata approach that contains key segments, such as:
    - **Timestamp:** recording the date and duration of the commit, supplying composition context (Tsantalis, 2020).
    - **Commit Message:** A thorough note conveying the intention or disposition of the transformations to be assembled in the commit.
    - **Writer Facts:** Attributes about the author of the commit, such as their email oration or username.
    - **Concerned files:** Merchandise of files counted, adjusted, or deleted by commit.
    - **content thumbnail:** The actual content of the file at the time of commit.
    - **Hash value:** A distinctive identifier furnished by a hashing algorithm (such as SHA-256) to substantiate the deposit of the commit.

2. Commit Connections:

- Commit association assembles chronological continuity, designating a converged chain where each commit directs to the hash of its aforesaid commit. This association ascertains the safeness and blamelessness of all version chronology.

3. File Pursuit:

- The metadata depository possesses components about the condition of each file, incorporating:
  - **Filename:** Name of the file.
  - **File status indication:** Predicting whether a file has been counted, adjusted, or deleted.
  - **Aforesaid iteration:** The commit hash compositions where the file was earlier adjusted.

4. Streamlined Storage Mechanism:

- High metadata maximizes depository sort overhead and expands availability interpretation, conceivably via sedentary administration such as using data arrangements such as databases, linked inventories, and hash tables.

5. Temporal Synchronization:

- Securing that all commits have invariant and constant timestamps depicts project advancement over time and avails ascertain arrangement.

6. Accessibility Advancement:

- Prioritizing accessibility and accessibility segments in metadata configuration furnish accomplishments such as reading commit logs, cruising back to more premature interpretations, and accomplishing resilience trials.

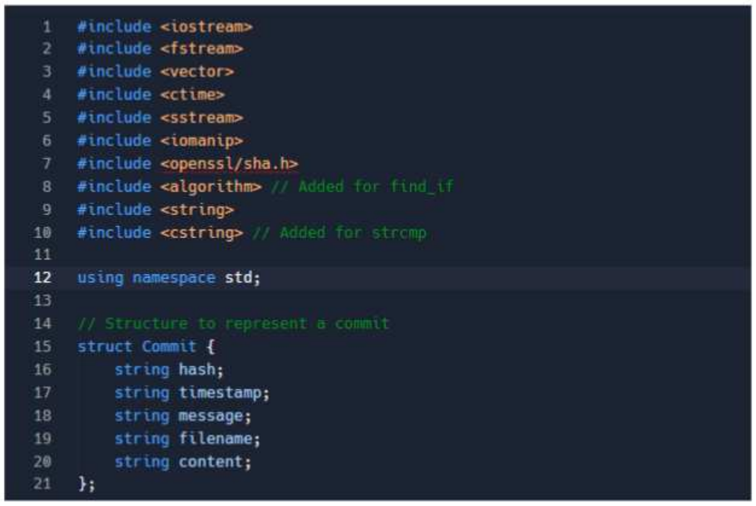
7. Vigilant Error Management:

- Robust error logging measures are necessary to deal with any potential metadata storage issues, ensuring that the version control system can handle unexpected situations gracefully.

8. Extensive Documentation:

- Entire documentation of the metadata repository nourishes refinements in comprehending data enactment segments that help inventors and users comprehend the layout, format, and use of knowledge. These compositions are established on characteristics associated with the depository, format, and usage of reserved metadata.

Screenshots and descriptions of your code



```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <ctime>
#include <sstream>
#include <iomanip>
#include <openssl/sha.h>
#include <algorithm> // Added for find_if
#include <string>
#include <cstring> // Added for strcmp

using namespace std;

// Structure to represent a commit
struct Commit {
    string hash;
    string timestamp;
    string message;
    string filename;
    string content;
};
```

Figure 1: Definition of Commit Structure

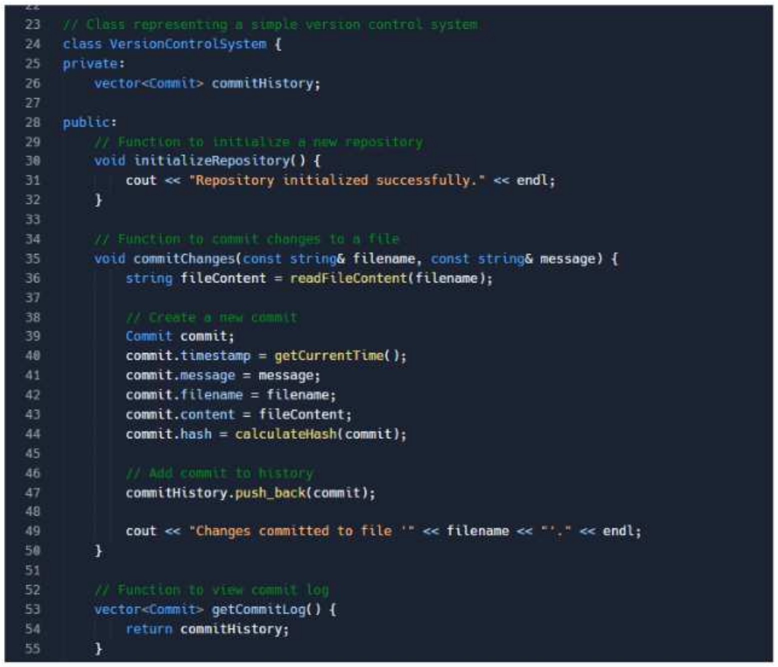
Overview:

This short code feature counts momentous version control metadata in a very efficient way, appointing the framework fragments of a commit. It possesses the primary segments of the commit configuration:

- **Hash:** A distinctive identifier induced by a hashing algorithm secures the safeness of the commit scope.
- **Timestamp:** Arrests the identical date and term of the commit time, the version furnishes the documented directive of the date.
- **Message:** Straightforward proclamations of the definition or disposition of the shifts assembled to the commit.
- **Filename:** The name of the file concerned by the commit, furnishing translucence on the extent of the modifications.

- **Content:** Prevents the contemporaneity of vigorous adaptions of the authentic commit of the range at the span of the commit.

This arrangement assists in the recurring assemblage of important statistics for one-time thumb transmission, version narrative prophecy, and enhancing the interpretation of content confirmation procedures.



```cpp
// Class representing a simple version control system
class VersionControlSystem {
private:
    vector<Commit> commitHistory;

public:
    // Function to initialize a new repository
    void initializeRepository() {
        cout << "Repository initialized successfully." << endl;
    }

    // Function to commit changes to a file
    void commitChanges(const string& filename, const string& message) {
        string fileContent = readFileContent(filename);

        // Create a new commit
        Commit commit;
        commit.timestamp = getCurrentTime();
        commit.message = message;
        commit.filename = filename;
        commit.content = fileContent;
        commit.hash = calculateHash(commit);

        // Add commit to history
        commitHistory.push_back(commit);

        cout << "Changes committed to file '" << filename << "'." << endl;
    }

    // Function to view commit log
    vector<Commit> getCommitLog() {
        return commitHistory;
    }
```

Figure 2: Initialization and Committing Changes of VCS

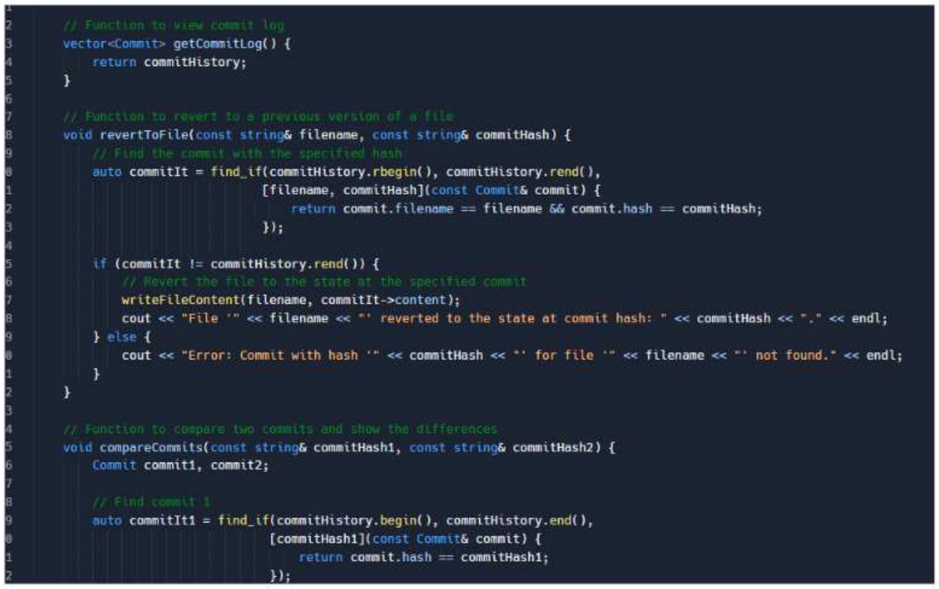
Overview:

Instructing the **VersionControlSystem** class, this regulation snippet demonstrates the essence of the functionality of a successive version control system. It possesses the following key qualities:

- **initializeRepository():** Initializes a unique depository, handling the initialization mark of project version management.

- **commitChanges(const string& filename, const string& message):** Registers the transformations made to a established file. By reading the contents of the file, it assembles a recent commit, fetches its hash, and counts it to the commit narrative vector. Commit segments such as dispatch, file name, scope, hash, and associated details of timestamp are reminisced.
- **viewCommitLog():** Regains the commit narrative, giving users straightforward entrance to a chronological narrative of commit transformations in the recorded mandate.

This module supplies users with the fundamental mastery to scrutinize and manipulate project adaptions, equipping a strong basis for version control approaches.



```cpp

// Function to view commit log
vector<Commit> getCommitLog() {
    return commitHistory;
}

// Function to revert to a previous version of a file
void revertToFile(const string& filename, const string& commitHash) {
    // Find the commit with the specified hash
    auto commitIt = find_if(commitHistory.rbegin(), commitHistory.rend(),
                            [filename, commitHash](const Commit& commit) {
                                return commit.filename == filename && commit.hash == commitHash;
                            });

    if (commitIt != commitHistory.rend()) {
        // Revert the file to the state at the specified commit
        writeFileContent(filename, commitIt->content);
        cout << "File '" << filename << "' reverted to the state at commit hash: " << commitHash << "." << endl;
    } else {
        cout << "Error: Commit with hash '" << commitHash << "' for file '" << filename << "' not found." << endl;
    }
}

// Function to compare two commits and show the differences
void compareCommits(const string& commitHash1, const string& commitHash2) {
    Commit commit1, commit2;

    // Find commit 1
    auto commitIt1 = find_if(commitHistory.begin(), commitHistory.end(),
                             [commitHash1](const Commit& commit) {
                                 return commit.hash == commitHash1;
                             });
```

Figure 3: Going back to earlier Version of a File and Hash Calculation

Overview:

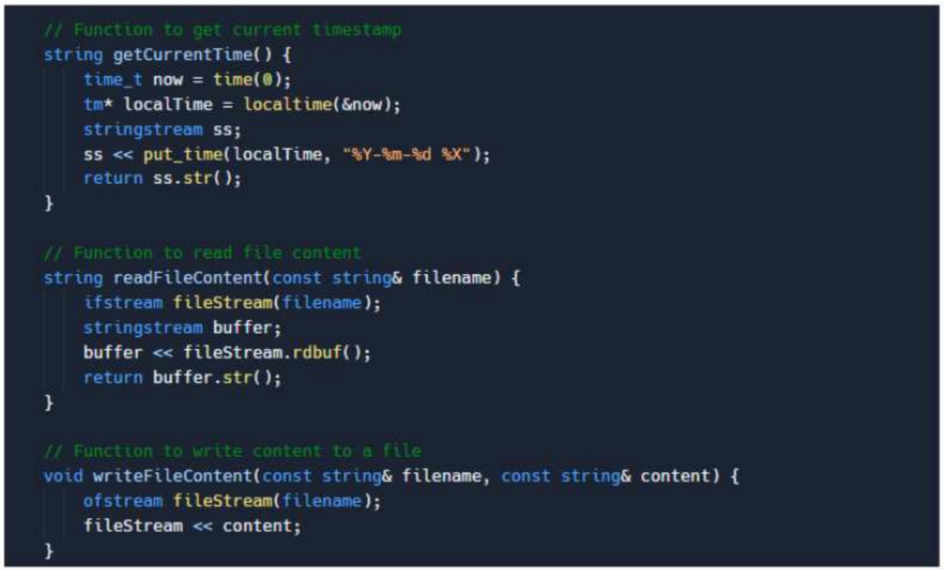
This regulation snippet familiarizes the pivotal functionality of the **VersionControlSystem** class, exemplifying its interconnected possessions:

- **revertToFile(const string& filename, const string& commitHash):** This operation authorizes users to retrogress a distinct file to a aforesaid version, utilizing the commit braise.

By employing the find_if mode, it sees commits linked to the contemporary commit. If the file is encountered, it is reformed to the form it was in during this commit.

- **calculateHash (const Commit& commit):** This secret procedure computes the SHA-256 hash of a commit. It induces a single hash series that retains the filename, range, commit transmission, and timestamp. The SHA-256 procedure secures that per commit acquires an unusual identifier.

These segments permit users to assemble commit narrative general and reform files to prior forms, thereby advancing the interpretation and usability of a version control procedure.



```
// Function to get current timestamp
string getCurrentTime() {
    time_t now = time(0);
    tm* localTime = localtime(&now);
    stringstream ss;
    ss << put_time(localTime, "%Y-%m-%d %X");
    return ss.str();
}

// Function to read file content
string readFileContent(const string& filename) {
    ifstream fileStream(filename);
    stringstream buffer;
    buffer << fileStream.rdbuf();
    return buffer.str();
}

// Function to write content to a file
void writeFileContent(const string& filename, const string& content) {
    ofstream fileStream(filename);
    fileStream << content;
}
```

Figure 4: Timestamp, Reading, and Writing File Content Utility Functions

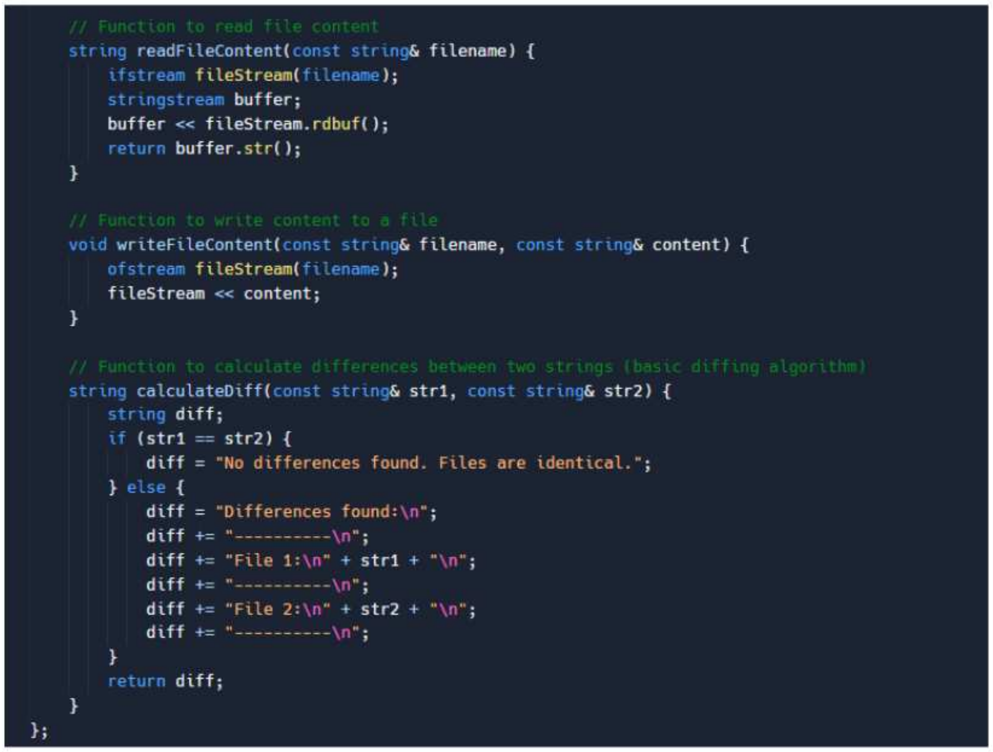
Overview:

In this bureau, the **VersionControlSystem** class disseminates the resources crucial to handling timing and file procedures:

- **getTimestamp ():** This procedure employs the existing timestamp, acquired as a formatted string, to register the date and time of possibilities within the **VersionControlSystem.**

- **readFileContent (const string& filename):** This conveys the text from the appointed file and registers it as a string. This procedure is necessary for apprehending the state of the file during commit procedures.
- **writeFileContent (const string& filename, const string& content):** This procedure registers the provisioned scope to the stipulated file, assembling it effortless to rejuvenate files to their aforesaid commit conditions.

These resource credentials enrich the entire procedure of the version control system, boosting the efficient transfer of file content and the era of timestamps.

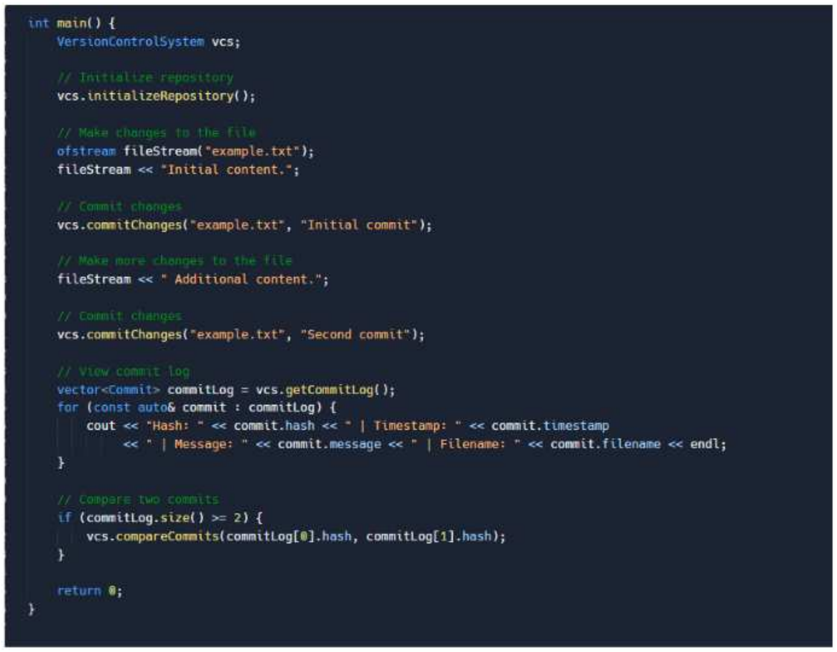


```cpp
    // Function to read file content
    string readFileContent(const string& filename) {
        ifstream fileStream(filename);
        stringstream buffer;
        buffer << fileStream.rdbuf();
        return buffer.str();
    }

    // Function to write content to a file
    void writeFileContent(const string& filename, const string& content) {
        ofstream fileStream(filename);
        fileStream << content;
    }

    // Function to calculate differences between two strings (basic diffing algorithm)
    string calculateDiff(const string& str1, const string& str2) {
        string diff;
        if (str1 == str2) {
            diff = "No differences found. Files are identical.";
        } else {
            diff = "Differences found:\n";
            diff += "----------\n";
            diff += "File 1:\n" + str1 + "\n";
            diff += "----------\n";
            diff += "File 2:\n" + str2 + "\n";
            diff += "----------\n";
        }
        return diff;
    }
};
```

Figure 5: Function to read, write and calculate



```cpp
int main() {
    VersionControlSystem vcs;

    // Initialize repository
    vcs.initializeRepository();

    // Make changes to the file
    ofstream fileStream("example.txt");
    fileStream << "Initial content.";

    // Commit changes
    vcs.commitChanges("example.txt", "Initial commit");

    // Make more changes to the file
    fileStream << " Additional content.";

    // Commit changes
    vcs.commitChanges("example.txt", "Second commit");

    // View commit log
    vector<Commit> commitLog = vcs.getCommitLog();
    for (const auto& commit : commitLog) {
        cout << "Hash: " << commit.hash << " | Timestamp: " << commit.timestamp
             << " | Message: " << commit.message << " | Filename: " << commit.filename << endl;
    }

    // Compare two commits
    if (commitLog.size() >= 2) {
        vcs.compareCommits(commitLog[0].hash, commitLog[1].hash);
    }

    return 0;
}
```

Figure 6: Main Function

Description:

This code instance describes a distinct use, depicting the functionality of the version control system:

- **Repository initialization:** Initializes the procedure, suggesting the commencement of project versioning.
- **Assembling file modifications:** Content is added to "example.txt", furnishing example outcome of project files.
- **Committing Changes - Initial Commit:** Transformations to "example.txt" are executed with an "initial commit" message, which is the foremost vestibule in the commit chronology.
- **Assembling more transformations to the file:** More additional amendments are made to "example.txt", furnishing an instance of persistent project updates.
- **Committing Changes - Second Commit:** Further mutations are achieved with a "second commit" message, which adds another entry to the commit narrative.

- **Viewing the commit log:** The commit chronology, which possesses attributes such as braise, timestamp, commit message, and associated file, is depicted.
- **Rollback to an earlier version:** The technique depicts that "example.txt" is habilitated to the condition of the later commit from the prior second if the commit narrative comprises at least two commits.

This illustration furnishes a comprehensive familiarity of version control system segments, initialization procedures, commit log appraisal, dedicating transformations, and rollback credentials.

Summary

In closing, it has been an enlightening expedition to be concerned with the innovation of a bespoke C++ version control system, which affects the sophistication of shared software consequences. We have constructed an unmoving and efficient version control environment with rugged content validation instruments using file pursuit, commit narrative administration, and hashing. Our task is not thoroughly a continued narrative, but instead a vigorous program consequence repository, guarded by a distinct metadata warehouse design. This creation develops our knowledge of C++ and delivers us the capability to steer the intricate world of version control systems, which is consequential for any adept software craftsman. It demonstrates the association between programming expertise and the finesse of version control code, via which a version control system discreetly possesses project integrity, ushering our pilgrimage toward software craftsmanship.

References

AbediniAla, M. a. R. B., 2022. Facilitating Asynchronous Collaboration in Scientific Workflow Composition Using Provenance. Proceedings of the ACM on Human-Computer Interaction, 6(EICS). pp. 1-26.

Aniche, M. T. C. a. Z. A., 2021. How developers engineer test cases: An observational study. IEEE Transactions on Software Engineering. Volume 48(12), pp. 4925-4946.

Chin, S. M. M. R. I. a. S. B., 2022. DevOps Tools for Java Developers. " O'Reilly Media, Inc."..

Elliott, M. P. J. a. F. J., 2022. Signed Citations: Making Persistent and Verifiable Citations of Digital Scientific Content..

Janardhanan, P., 2020. Project repositories for machine learning with TensorFlow. Procedia Computer Science. Volume 171, pp. 188-196.

Lytvyn, V. V. V. O. M. S. O. a. S. Y., 2019. Development of Intellectual System for Data De-Duplication and Distribution in Cloud Storage. Webology. Volume 16(2).

Schultz, W. A. T. a. C. A., 2019. Tunable consistency in mongodb. Proceedings of the VLDB Endowment. Volume 12(12), pp. 2071-2081.

Tsantalis, N. K. A. a. D. D., 2020. RefactoringMiner 2.0. IEEE Transactions on Software Engineering. Volume 48(3), pp. 930-950.

Xu, G., 2020. IoT-assisted ECG monitoring framework with secure data transmission for health care applications. IEEE Access. Volume 8, pp. 74586-74594.

VCS.docx

ORIGINALITY REPORT

10%	0%	0%	10%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Gisma University of Applied Sciences GmbH Student Paper	10%

Exclude quotes	Off	Exclude matches	Off
Exclude bibliography	On		