Breakdown of the VCS code

"**git_utils.h**" is a header that contains declarations for various functions and constants related to operations. Functions for compressing and decompressing objects and files through operations such as decompress_object, decompress_file, compress_str, and compress_to_file

Main components:

init: Initializes a Git repository in a specified directory

cat_file: Displays the contents of a Git object

read_tree: Reads the contents of a Git tree object

write_tree: Creates a tree object from a directory

restore_tree: Restores a tree object to a specified destination

create_blob: Creates a blob object from a file

hex_to_binary: Converts hexadecimal to binary

sha1: Computes SHA1 hash of data

digest_to_hash: Converts a digest to a hash string

commit_tree: Creates a commit object

clone: Implements Git clone functionality

curl_request: Performs HTTP requests for remote repository operations

“**blob.cpp”** implements a function that takes a file path as input and attempts to open the file in binary mode. If the file can't be opened, it prints an error message and exits the program.

Main components:

The function reads the entire content of the file into a string.

It creates a header string in the format "blob <content_size>\0", where <content_size> is the length of the file content.

The header and content are combined to form the "store" string.

A SHA1 hash is calculated for the store string, which becomes the blob's identifier.

The function creates a directory structure based on the first two characters of the SHA1 hash.

The blob data is compressed using zlib.

The compressed data is then written to a file in the created directory structure, with the filename being the remaining characters of the SHA1 hash.

The function returns the full SHA1 hash of the blob in the end.

“**cat_file.cpp”** implements a function that takes command-line arguments, expecting at least 3 arguments.

Main components:

It checks if the second argument (flag) is "-p", which is used to print the contents of an object.

The function then processes the third argument as a hash, which represents the object to be displayed.

It constructs a file path based on the hash, following Git's object storage structure (.git/objects/xx/yyyyy...).

The function attempts to open and read the file at the constructed path.

If successful, it decompresses the object data using a decompress_object function.

After decompression, it trims the metadata (everything before the first null byte) and prints the remaining content.

The function handles potential errors, such as invalid arguments, file opening failures, or filesystem errors.

“clone.cpp” implements basic clone fuctionality.

Main components:

It defines functions for handling pack files, which are used to efficiently transfer repository data.

It includes a function `apply_delta` that applies delta compression to reconstruct full objects from base objects and delta information.

The main `clone` function is responsible for cloning a repository:
- Checks for proper URL and optional directory.
- Creates the target directory for the clone.
- Initializes a new Git repository in the target directory.
- Retrieves the pack file from the remote repository using a `curl_request` function

It then processes the pack file:
- It reads the number of objects in the pack.
- Iterates through each object, handling different types (commit, tree, blob).
- Reconstructs delta-compressed objects using the `apply_delta` function.
- Decompresses and stores objects in the local repository.

It identifies the master commit contents and uses them to restore the working tree.

It handles both regular objects and delta-compressed objects, storing them in the appropriate format in the local `.git/objects` directory.

It restores the working tree based on the master commit, effectively completing the clone operation.

“commit.cpp” implements a function called commit_tree that creates a commit object.

Main components:

It takes command-line arguments, expecting at least 6 arguments.

It extracts the following information from the arguments; Tree SHA, Parent commit SHA and the commit message.

It constructs the commit content, including tree reference, parent commit reference, author and committer information and the commit message.

It creates a header for the commit object, which includes the object type ("commit") and its size.

It then combines the header and content to form the complete commit object.

The SHA1 hash of the commit object is calculated, which becomes the commit's unique identifier.

The function creates a directory structure in the .git/objects folder based on the first two characters of the SHA1 hash.

The commit object is compressed using zlib compression.

The compressed data is written to a file in the .git/objects directory, with the filename being the remaining characters of the SHA1 hash.

The function then gives the full SHA1 hash of the commit as output.

“compress.cpp” implements compression and decompression functionality.

Main components:

File Compression and Decompression:
- compress_file: Compresses the contents of an input file and writes the compressed data to an output file using zlib's deflate algorithm.
- decompress_file: Decompresses data from an input file and writes the decompressed content to an output file using zlib's inflate algorithm.

String Compression and Decompression:
- compress_str: Compresses a given string using zlib's deflate algorithm.
- decompress_str: Decompresses a compressed string using zlib's inflate algorithm.

Git-specific Functionality:

- compress_to_file: This function is designed to compress content and store it in the Git object database. It creates the necessary directory structure (.git/objects/xx/...) and saves the compressed content as a Git object.
- decompress_object: Decompresses a Git object stored in the compressed format.

Error Handling:
The code includes extensive error checking and reporting for various scenarios such as file I/O errors, compression/decompression failures, and memory allocation issues.

Utility Functions:
- The code uses helper functions and structures from zlib (z_stream, inflate, deflate) to perform the actual compression and decompression operations.

File System Operations:
- It uses C++ filesystem functions to create directories and check for file existence when storing Git objects.

“curl_utils.cpp” implements functionality for interacting with repositories using libcurl, a library for making HTTP requests.

Main components:

It defines two callback functions:
- write_callback: Used to process the initial response from the Git server, extracting the master branch hash.
- pack_data_callback: Accumulates received data into a string.

The main function curl_request performs two HTTP requests to interact with a Git repository:
- First request: It fetches the references (refs) from the repository by appending "/info/refs?service=git-upload-pack" to the provided URL. This request retrieves the hash of the master branch.
- Second request: It sends a POST request to "/git-upload-pack" endpoint, requesting the pack data for the master branch. This request includes the previously obtained hash in the post data.

It uses libcurl to set up and execute these HTTP requests:
- It initializes a CURL handle and sets various options like URL, callback functions, and headers.

- For the second request, it sets a custom Content-Type header for the git-upload-pack request.

After each request, the received data is processed using the respective callback functions.

Finally, the function cleans up the CURL resources and returns a pair containing:
- The pack data received from the second request
- The hash of the master branch obtained from the first request

“init.cpp” implements the initialization process for a repository.

Main components:

It creates a new directory structure for a repository:
- Creates a .git directory
- Creates subdirectories .git/objects and .git/refs

It creates a .git/HEAD file and writes the initial content:
- The HEAD file is set to point to refs/heads/main, indicating the default branch name

The function handles potential errors:
- It uses a try-catch block to handle filesystem errors
- If it fails to create the HEAD file, it prints an error message

Upon successful initialization, it prints "Initialized git directory"

The function returns true if the initialization is successful, and false if any errors occur

“server.cpp” implements a command-line application that handles various version control operations.

Main components:

The program accepts command-line arguments to determine which operation to perform.

It supports the following commands:

- init: Initializes a new repository.
- cat-file: Displays the contents of a Git object.
- hash-object: Creates a blob object from a file and stores it in the repository.
- ls-tree: Lists the contents of a tree object.
- write-tree: Creates a tree object representing the current directory structure.
- commit-tree: Creates a commit object.
- clone: Clones a repository.

For each command, the program calls a corresponding function to perform the operation.

The code includes error handling for invalid commands or insufficient arguments.

It uses custom functions defined in a separate "git_utils.h" header file.

The program returns EXIT_SUCCESS (0) if the operation is successful, and EXIT_FAILURE (1) if an error occurs.

“sha1_utils.cpp” provides utility functions for working with SHA-1 hashes and hexadecimal representations.

Main components:

The hex_to_binary function converts a hexadecimal string to its binary representation.

It processes the input string two characters at a time, converting each pair to its corresponding byte value.

The digest_to_hash function takes a digest (a binary representation of a hash) and converts it to a hexadecimal string representation.

It uses a stringstream to format each byte of the digest as a two-character hexadecimal value.

The hash function computes the SHA-1 hash of a given input string.

It uses the OpenSSL library's SHA1 function to calculate the hash and then converts the resulting digest to a hexadecimal string representation.

The SHA-1 hash function, in particular, is used to generate unique identifiers for commits, trees, and blobs in Git's object database.

“tree.cpp” implements several functions for Git-like tree operations.

Main components:

read_tree: This function reads a tree object from the Git-like object store. It takes command-line arguments, including an optional "--name-only" flag. The function performs the following tasks:
- Validates input arguments
- Reads the tree object from the specified path
- Decompresses the object data
- Parses the tree content, extracting file and directory names
- Sorts and prints the names of files and directories in the tree

write_tree: This function creates a tree object from a given directory. It recursively traverses the directory structure, creating blob objects for files and nested tree objects for subdirectories. The function:
- Collects entries (files and directories) in the given directory
- Sorts the entries
- Creates a tree object with the sorted entries
- Compresses and stores the tree object in the Git-like object store
- Returns the SHA1 hash of the created tree object

restore_tree: This function restores a tree object to the filesystem. It takes a tree object hash, a destination path, and the repository directory as input. The function:
- Reads and decompresses the tree object
- Parses the tree contents
- Recursively creates directories and files based on the tree structure
- For files, it decompresses and writes the corresponding blob objects to the filesystem