1. Content tracking is the sequence of how documents and code are revised as far as what is contained, and corrected, in them, whereas context tracking is how integrators and collaborators merge those documents to form meaningful project content. This entire chapter provides good examples of content tracking. The hallmark of context tracking is the improvement in the performance of a system of code, reflected in its revision history as shipped to a user base, or customers. Context tracking focuses on tracking changes based on the context or structure of the file. Instead of analyzing content at a granular level, it identifies changes based on higher-level structural differences, such as lines added, removed, or modified in a file, and why that is done.

2. At least for software coding projects, an integrator oversees the merging of content, by either accepting or rejecting pull requests, based upon the integrators overview of everything in the project.

3. Use the **rm -r** command to delete the entire directory containing the project files.

4. The  **git diff** command in Git is similar in concept to the **diff** command in Linux. Both commands are used to compare and display differences between files. The **git diff** command is primarily used to show changes between commits, branches, or the working directory in a Git repository. The **diff** command in Linux is a general-purpose utility that compares the content of two files line by line and displays the differences in a user-readable format. It is not tied to version control systems like Git, and is used for comparing arbitrary files.

5. Aside from source code files-

*a. Build Configuration Files:*

Files that specify build configurations, such as Makefile or build scripts (build.sh, build.bat). These define how to compile the source code into an executable or library.

*b. Build Output and Artifacts:*

Compiled binaries (e.g., executables, object files) and generated build artifacts should not be stored in the repository. However, you might want to include a .gitignore file to ensure Git ignores these files during commits.

*c. Build Logs:*

Logs generated during the build process can be helpful for troubleshooting. You may choose to track or exclude them based on your project's requirements.

*d. Dependency Files:*

Files related to project dependencies, like package.json, CMakeLists.txt, or configure.ac. These files specify the dependencies and versions required for building the project.

*e. Documentation:*

Documentation files (e.g., README.md, user manuals) that provide information about the project, usage, and development guidelines. It's crucial to track and maintain these files for reference and clarity.

*f. Configuration Files:*

Configuration files that define project settings, environment variables, or any other configuration necessary for building or running the application.

*g. Test Scripts and Data:*

Scripts or data used for testing the C program, such as test cases, testing frameworks, or input/output data. These can be staged for version control to ensure consistent testing procedures.

*h. Scripts and Automation:*

Scripts that automate repetitive tasks, set up the development environment, or handle other automation aspects. These can be beneficial to track for consistent development workflows.

6. No answer required.

7. No answer required.

8.To retrieve an earlier commit and apply its changes to the working directory, you can use the **git checkout** command followed by the commit hash, or a reference to that commit.

9. No answer required.

10. No answer required.

11. To see an abbreviated list of commits in the current branch along with their commit messages, you can use the **git log** command with appropriate options. The **--oneline** option is commonly used to display a concise list of commits with their abbreviated hash and commit messages. For example- **git log --oneline**

12. **curl -s https://api.github.com/repos/bobk48/RaspberryPiOS** for the refspec URL.

From the web interface, the Activity button under About for the fetch assignments

or **git fetch origin master** from the command line.

13. a. Yes, you can drag and drop whole directories into a repository via the web browser interface to GitHub.

b. This can be done as follows:

b.1. Create a directory on the local Raspberry Pi to contain the GitHub directory.

Make that directory the current working directory.

b.2. Use the **git clone** command to clone the new GitHub directory to the local directory, such as-

**git clone https://github.com/bobk48/interactive**

b.3. Add some files to this cloned repository locally, or edit existing ones in it, if there are any,

like a README.md file that you might have added to it as part of the creation of **interactive**.

b.4. Commit those changes.

b.5. Use the **git push** command to push your changes from the local repository to the GitHub one.

You’ll have to supply your username and PAT in order to authenticate yourself to GitHub

when using the **git push** command.

b.6. Also, be sure you’re pushing the changes to the correct branch of the new GitHub

directory, such as-

**git push https://github.com/bobk48/interactive main**

Not the branch master!

These are some of the reasons you’d want to do this-

**Version Control:** Git provides version control capabilities, allowing you to track changes to files over time. This is essential for collaboration and managing the development process effectively.

**History and Tracking:** With Git, you can track the history of changes, understand who made those changes, and revert to previous versions if needed. This is crucial for maintaining a reliable and organized development workflow.

**Collaboration and Concurrent Work:** Git enables collaboration among multiple developers. By using Git commands, you can efficiently handle concurrent work on the repository and merge changes made by different contributors.

**Branching and Merging:** Git allows you to create branches for new features, bug fixes, or experiments without affecting the main codebase. You can then merge these changes back into the main branch, ensuring a controlled integration process.

**Conflict Resolution:** Git helps manage conflicts that may arise when multiple contributors modify the same files. It provides tools to resolve conflicts in an organized and systematic manner.

**Automation and Scripting:** Git commands can be automated and scripted, allowing for more efficient and consistent management of the repository. This is especially beneficial for repetitive tasks or processes that require specific configurations.

**Integration with Tools and CI/CD:** Git integrates seamlessly with various development tools and continuous integration/continuous deployment (CI/CD) pipelines, enabling a streamlined development workflow and automation.

**Advanced Operations:** Git offers a wide range of advanced operations and functionalities, such as rebasing, cherry-picking, tagging, and more, which can be utilized to manage the repository in a more sophisticated and controlled manner.

In summary, using Git commands to manipulate files in a GitHub repository, even after initially adding them via drag and drop in a web browser, provides a structured and efficient way to manage the development process, collaborate with team members, track changes, and maintain a reliable version control system.

14. Descend to the file you want to delete, and click on the icon to the extreme right that looks like three dots (. . .), and make the Delete file choice at the bottom of the pull-down menu presented. Confirm the deletion by making the Commit button choice to delete that file.

15. Go to your repository, and click on the Settings icon( it looks like a gear). Scroll down to the bottom of the Settings page, to the Danger Zone. Click on the icon shown in red, Delete this repository. Make the multiple confirmation choices under Delete this repository to delete the selected repository.