Lecture 11: Version Control Continued

1. Introduction

This lecture continues the exploration of Git as a case study in version control, focusing on the workflow, commit lifecycle, file states, and practical commands. Emphasis is placed on understanding the relationship between the working directory, the index (staging area), and the object database (history), as well as the importance of commit messages, commit IDs, and repository configuration.

---

2. Git's Three States: Working Directory, Index, and Object Database

- **Working Directory (Working Files):**
  - The files you actively edit. These are visible to compilers, test programs, and other tools.
  - Changes here are not yet tracked by Git until staged.
- **Index (Cache/Staging Area/Planned Future):**
  - Where you stage files for the next commit using `git add`.
  - Represents your "planned future"—what you intend to commit next.
  - You can stage multiple changes before committing.
- **Object Database (History):**
  - Contains all committed snapshots (commits) of your project.
  - Commits are immutable; you cannot change history, only add new commits.

ASCII Diagram: Git State Transitions

```
[Working Directory] --git add--> [Index (Staging Area)] --git commit--> [Object Database (History)]
      ^                              |                                   |
      |                              |                                   |
      +-----------git checkout-------+                                   |
      |                                                                  |
      +----------------------------git reset--------------------------+
```

Table: Comparing Git States

State	Description	Typical Commands
Working Directory	Files you edit directly	`git status`, edit files
Index (Staging)	Planned changes for next commit	`git add`, `git reset`
Object Database	All committed snapshots (history)	`git commit`, `git log`

Example: File Lifecycle

Suppose you have a file `hello.txt`:

```
echo "Hello" > hello.txt
git status
# hello.txt is untracked

git add hello.txt
git status
# hello.txt is staged

git commit -m "Add hello.txt"
git status
# Working directory clean
```

State	File Status	Command to Move to Next State
Working Directory	Untracked/Modified	`git add <file>`
Index (Staging Area)	Staged	`git commit`

State	File Status	Command to Move to Next State
Object Database	Committed	(N/A, file is now in history)

3. Comparing File States with `git diff`

Command	Compares	Description
`git diff`	Working directory vs. staged (index)	Shows changes not yet staged
`git diff --cached`	Staged (index) vs. last commit (HEAD)	Shows what is staged but not yet committed
`git diff HEAD`	Working directory vs. last commit	Shows all changes since last commit
`git diff <commit>`	Working directory vs. specific commit	Compare with any previous commit

- `HEAD` is a symbolic name for the most recent commit.
- Use `git log` to view commit history and obtain commit hashes.

Example: `git diff` Variants

Suppose you edit `hello.txt` after committing:

```
echo "World" >> hello.txt
git add hello.txt
echo "!" >> hello.txt
```

Command	What it shows	Example Output (abbreviated)
`git diff`	Changes in working directory not staged	Shows addition of "!"
`git diff --cached`	Changes staged for commit	Shows addition of "World"
`git diff HEAD`	All changes since last commit	Shows both "World" and "!"

4. Staging and Committing Changes

- **Staging:**
  - `git add <filename>` stages a file for the next commit.
  - You can stage multiple files before committing.
- **Committing:**
  - `git commit` opens an editor for a commit message.
  - The commit message should explain *why* the change was made, not just *what* changed.
  - Good commit messages are crucial for future understanding and collaboration.
  - Some projects enforce commit message formats using hooks (e.g., Diffutils).
- **Hooks and Commit Rejection:**
  - Projects may use hooks to enforce commit message standards.
  - If your message doesn't match, the commit may be rejected.

Example Workflow

```
# Edit file
echo "Edit" >> file.txt

# Stage file
git add file.txt

# Commit staged changes
git commit -m "Describe the edit"
```

Good Commit Message Example

```
Add user authentication to login page

- Implemented password hashing
- Added login error messages
- Updated user model
```

---

5. Commit IDs and SHA-1 Checksums

- **Commit IDs:**
  - Each commit is identified by a SHA-1 hash (160 bits, 40 hex characters).
  - The hash is computed from the commit's contents and metadata (author, date, message, parent commit(s)).
  - Changing any part of a commit results in a new hash.
- **Immutability:**
  - Commits are immutable; you cannot change a commit in place.
  - If you change a commit, a new commit with a new hash is created.
- **Collision Probability:**
  - The chance of two different commits having the same SHA-1 hash is astronomically low.
- **Why Not Sequential IDs?**
  - Sequential IDs require a central server and are problematic for distributed systems.
  - SHA-1 allows decentralized, parallel development without conflicts.

Real-World Analogy: SHA-1 Commit IDs

Think of a commit as a unique fingerprint for a snapshot of your project. If you change even a single character in a file, the fingerprint (SHA-1 hash) changes completely. This ensures that every version is uniquely identified and tamper-evident, much like a tamper-proof seal on a document.

How a Commit Hash is Computed

A commit hash is generated from:

- The contents of the commit (tree, parent, author, date, message)
- Example (simplified):

```
commit 1a2b3c4d...
tree 9e8f7g6h...
parent 0z9y8x7w...
author Alice <alice@example.com> 1712345678 -0700
committer Alice <alice@example.com> 1712345678 -0700

Add feature X
```

Changing any part (even a space in the message) changes the hash.

---

6. Amending Commits

- `git commit --amend`:
  - Allows you to replace the most recent commit with a new one (e.g., to fix mistakes or update the message).
  - The amended commit gets a new SHA-1 hash.
  - Use with caution; beginners are encouraged to let history accumulate naturally.

Example: Amending a Commit

```
git commit -m "Initial commit"
# Realize you forgot a file
git add forgotten.txt
git commit --amend
```

Before:

```
* 1234abcd Initial commit
```

After:

```
* 5678efgh Initial commit (now includes forgotten.txt)
```

Warning: Never amend commits that have been pushed/shared with others.

---

7. Convenience Features and Shortcuts

- **Committing All Changes:**
  - `git commit -a -m "message"` stages and commits all tracked, modified files with a single command.
  - Not recommended for beginners; be explicit about what you stage.
- **Viewing Tracked Files:**
  - `git ls-files` lists all files tracked by Git in the current commit.
- **Searching Content:**
  - `git grep <pattern>` searches for a pattern in all tracked files.
  - Equivalent to `git ls-files | xargs grep <pattern>` but more efficient.

---

8. `git status` and Untracked Files

- `git status` shows:
  - Modified files
  - Staged files
  - Untracked files (not in Git or `.gitignore`)
- **Untracked Files:**
  - Git uses heuristics and `.gitignore` to decide which files to show as untracked.

---

9. Ignoring Files with `.gitignore`

- **Purpose:**
  - `.gitignore` lists patterns for files Git should ignore (e.g., build artifacts, temporary files).
- **Pattern Examples:**

Pattern	Meaning
`*.o`	Ignore all object files in any directory
`/main.mk`	Ignore only at repository root
`build/`	Ignore the entire `build` directory
`/gnu-lib-test/`	Ignore the `gnu-lib-test` directory at root

- **Multiple `.gitignore` Files:**
  - You can have `.gitignore` in any directory; Git checks the closest one first.
- **Versioning:**
  - `.gitignore` is usually versioned and shared with collaborators.

Example `.gitignore` File

```
# Ignore all .log files
*.log

# Ignore build/ directory everywhere
```

```
build/

# Ignore only at root
/.env

# Ignore node_modules in any subdirectory
node_modules/
```

Pattern	Matches Example	Does Not Match Example
`*.log`	`debug.log`, `foo/bar.log`	`foo/bar.txt`
`build/`	`build/`, `foo/build/`	`builder/`
`/.env`	`.env` at root	`foo/.env`

---

10. Git Repository Configuration

- **`.git/config`:**
  - Stores repository-specific settings (core options, remotes, branches, submodules).
  - Not under version control.
- **Viewing and Editing:**
  - `git config -l` lists all configuration settings.
  - `git config <key> <value>` sets a configuration value.
  - You can edit `.git/config` directly, but back it up first.

---

11. Commit Reference Syntax and Arithmetic

- **Shorthand Notation:**

Syntax	Meaning
`HEAD`	Latest commit on current branch
`HEAD^`	First parent (one commit back)
`HEAD^^`	Two commits back
`HEAD~3`	Three commits back
`commit^2`	Second parent (for merge commits)
`HEAD^!`	Difference between current commit and its parent

- **Tags:**
  - Tags can be used as human-readable names for commits.
  - Use `--` to separate commit references from file names to avoid ambiguity.

---

12. `git blame`

- **Purpose:**
  - Shows the last commit that modified each line of a file, along with author and date.
- **Usage:**
  - `git blame <filename>`
- **Cautions:**
  - Not always reliable for finding the true origin of a bug (may only show formatting changes).
  - Should not be used to assign blame in a punitive sense; can create a toxic environment.
  - Use to trace history, not to punish.
- **Tracing Bugs:**
  1. Use `git blame` to find the commit that last changed a line.
  2. Use `git diff <commit>^!` to see what that commit changed.

---

13. Summary Table: Key Git Commands

Command	Description
`git add <file>`	Stage file for commit
`git commit`	Commit staged changes
`git commit --amend`	Amend the last commit
`git status`	Show working directory status
`git diff`	Show unstaged changes
`git diff --cached`	Show staged changes
`git log`	Show commit history
`git ls-files`	List tracked files
`git grep <pattern>`	Search tracked files for a pattern
`git blame <file>`	Show last commit for each line in a file
`git config -l`	List configuration settings

---

14. ASCII Diagram: Git Commit Lifecycle

```
[Working Directory]
      |
      v   git add
   [Index / Staging Area]
      |
      v   git commit
   [Object Database / History]
```

---

15. Best Practices and Warnings

- Write meaningful commit messages explaining *why* changes were made.
- Don't obsess over perfect history; it's okay to make mistakes and fix them in later commits.
- Use `.gitignore` to avoid tracking unnecessary files.
- Be careful when editing `.git/config` directly.
- Use `git blame` for understanding history, not for assigning fault.
- Prefer explicit staging (`git add`) over `git commit -a` for clarity.

Checklist: Git Best Practices

Do's	Don'ts
Write clear, descriptive commit messages	Use vague messages like "fix"
Stage files explicitly with `git add`	Rely on `git commit -a` for everything
Use `.gitignore` for unneeded files	Commit build artifacts
Rebase only local, unpublished branches	Rebase shared/public branches
Use `git status` and `git log` frequently	Edit `.git/config` without backup
Resolve merge conflicts carefully	Use `git blame` to assign fault

---

16. Branching

A **branch** is a lightweight, movable pointer to a commit. Branches help isolate development work and enable parallel feature development, bug fixes, and experimentation.

16.1 Common Use Cases

Type	Purpose
Maintenance	Stable bugfix-only version
Feature	Work on specific new features

Type	Purpose
Experimental/Fork	Alternative implementations or separate project directions

16.2 Managing Branches

```
git branch                # list branches
git branch newbranch      # create a new branch
git checkout newbranch    # switch to a branch
git checkout -b newbranch commit  # create and switch
git branch -d name        # delete a branch (warns if commit is unmerged)
git branch -D name        # force delete
git branch -m old new     # rename branch
```

Example: Branch Workflow

```
git branch feature-x         # Create branch
git checkout feature-x       # Switch to branch
# ... make changes ...
git commit -am "Work on feature X"
git checkout master
git merge feature-x          # Merge changes into master
git branch -d feature-x      # Delete branch
```

Table: Branch Commands

Command	Description
`git branch`	List branches
`git branch <name>`	Create new branch
`git checkout <name>`	Switch to branch
`git checkout -b <name>`	Create and switch to new branch
`git branch -d <name>`	Delete branch (safe)
`git branch -D <name>`	Delete branch (force)
`git branch -m <old> <new>`	Rename branch

16.3 Visual Representation

```
            A---B---C (master)
                    \
                     D---E---F (feature)
```

- `master` and `feature` point to different commits.
- Use `git checkout` to switch between them.
- Commits form a Directed Acyclic Graph (DAG), where arrows point backwards to parent commits.

---

17. Merging

Merging combines content from two different branches into a single commit.

```
git merge branchname
```

17.1 Merge Commit

- When merging master with another branch, Git auto-generates a new commit containing both histories.
- If there's no conflict:
  - Git auto-merges changes.
  - Creates a new commit with lineage from both parents.

17.2 Merge Conflicts

- Occur when the same lines are changed in both branches.
- Git inserts markers like:

```
<<<<<<< HEAD
old version
=======
new version from other branch
>>>>>>> branchname
```

- You must manually resolve conflicts and commit the result.

Example: Merge Conflict and Resolution

Suppose both `master` and `feature` modify the same line in `file.txt`.

```
git checkout master
echo "Line from master" > file.txt
git commit -am "Master change"

git checkout feature
echo "Line from feature" > file.txt
git commit -am "Feature change"

git checkout master
git merge feature
```

Result:

```
<<<<<<< HEAD
Line from master
=======
Line from feature
>>>>>>> feature
```

Resolution:
Edit the file to resolve the conflict, then:

```
git add file.txt
git commit
```

17.3 Merge Algorithms

- Git identifies the "closest common ancestor" (e.g., commit A).
- Computes changes from A to B (`diff B A`) and A to C (`diff C A`).
- If non-overlapping, Git applies changes cleanly.
- Overlapping changes cause a conflict requiring manual intervention.

---

18. Rebasing

Rebasing re-applies commits from one branch to another base commit.

```
git rebase branchname
```

- Instead of creating a merge commit, rebasing rewrites history to make it appear as if your changes were applied on top of the target branch.

18.1 Visual Comparison

Merge:

```
    A---B---C---M (merged)
         \     /
          D---E
```

Rebase:

```
    A---B---C---D'---E' (rebased)
         \
          D---E
```

- Rebase creates new commits (`D'`, `E'`) with same changes, but different parent.
- Resulting history is "clean", linear.

Table: Merge vs. Rebase

Feature	Merge	Rebase
History	Preserves all branches and merges	Linearizes history
Commit Hashes	Unchanged for existing commits	New hashes for rebased commits
Use Case	Collaborative, public branches	Local, private branches
Conflict Resolution	May need to resolve once	May need to resolve at each commit

18.2 When to Use

- Clean, linear history after local development before pushing.
- Avoid publish-rebase: rewriting shared history causes complications.

---

19. Best Practices

- **Commit Messages:**
  - Describe reasoning, not just the change.
  - Avoid generic messages like "xyz."
- **Avoid Perfectionism:**
  - Don't rely heavily on `--amend` early on.
  - Mistakes are natural and traceable; history should reflect development flow.
- **Merging vs. Rebasing:**
  - Merging preserves actual development paths.
  - Rebasing simplifies history—useful for polishing before sharing.

---

20. Additional ASCII Diagram: Git Workflow

```
[Working Directory]
      |
      v   git add
   [Index / Staging Area]
      |
      v   git commit
   [Object Database / History]
      |
      v   git branch
   [Branch Pointer]
      |
      v   git merge / git rebase
   [Integrated History]
```