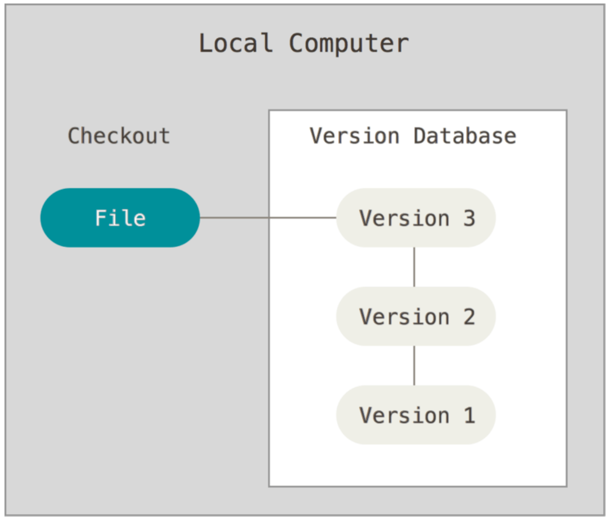
Git basics

James O’Neill
Atmospheric Processes Group Meeting
14/12/16

Key source: https://git-scm.com/doc

git vs other VCSs

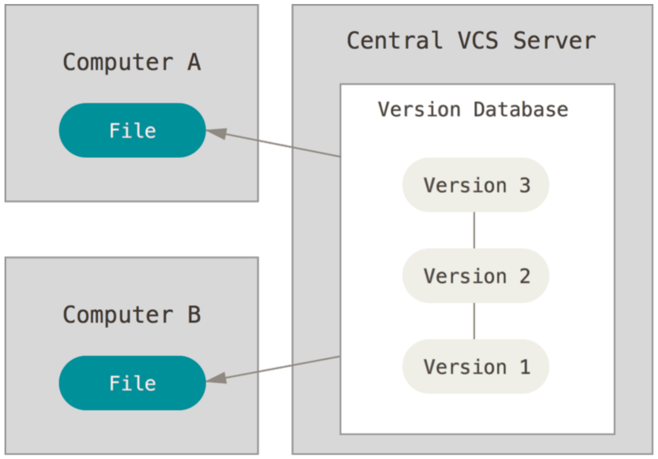
Local VCSs
e.g. RCS



Pros: Better than nothing…

Cons: High risk of data loss; collaboration impractical

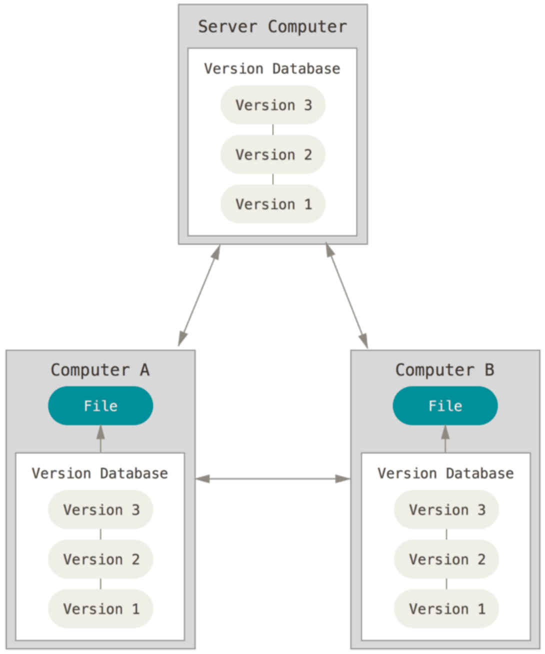
Centralised VCSs
e.g. SVN, CVS, Perforce



Pros: Collaboration possible; lower risk of (present state) data loss

Cons: Server downtime prevents work; High risk of historical data loss

Distributed VCSs
e.g. git, Mercurial, BitKeeper

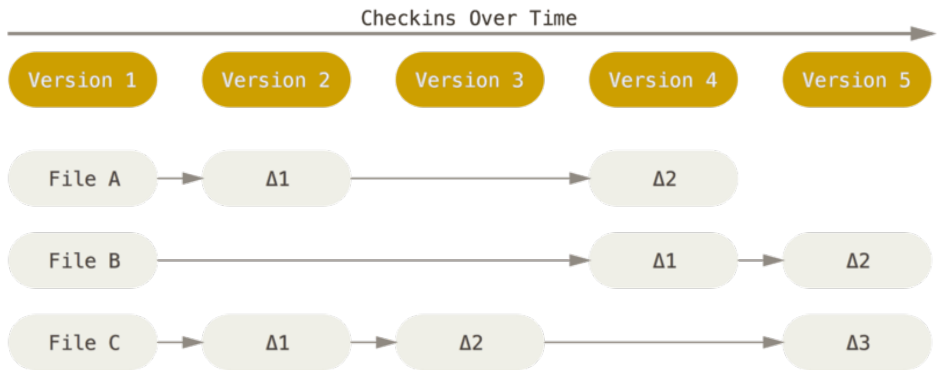


Pros: Very low risk of any data loss; multiple remote repositories possible

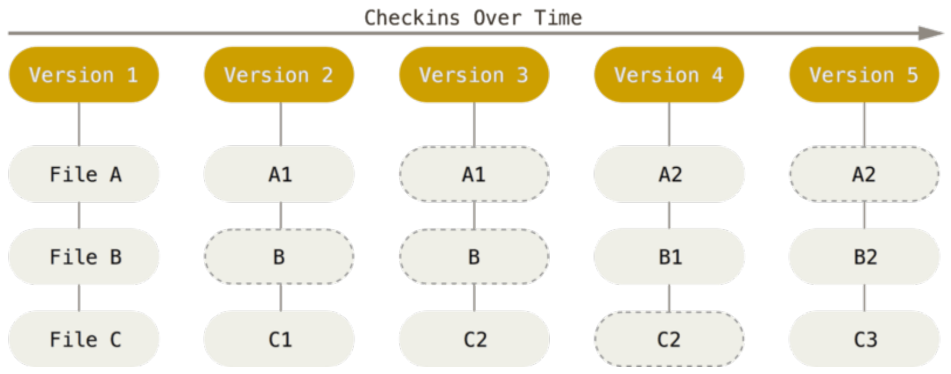
Cons: Requries more diskspace

git: think commit

Most other VCSs



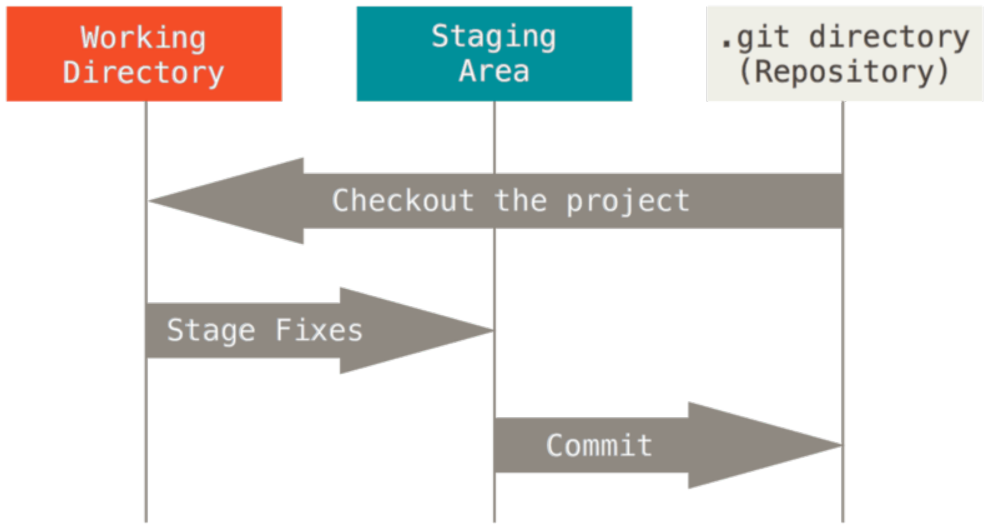
git



- Git stores versions as snapshots (or 'commits') of the entire project over time, rather than as patches
- A new version is created every time a user commits their changes
- For memory efficiency, unchanged files are stored as links to the previous file
- Each commit has an associated commit-id – a 40-character hex code, e.g. de31c72f16a5c8ec33ce9acbea6a0cd1ab2d77f9

Local file-system

- All version data and metadata are stored is stored in the (hidden) .git sub-directory
- The main ('working') directory initially contains all the files from the most recent snapshot (ignoring branches for now)
- After modifying files, add those you want to commit to the staging area
- Perform a commit → snapshot of all staged files and previous versions of other files permanently stored in the git history



Initial git setup and new repository

- Ensure git is installed (On Ubuntu: `sudo apt-get install git-all`)
- Set username and email address once (won’t be able to commit otherwise):

```
$ git config --global user.name "Jonny Git"
$ git config --global user.email j.git@gmail.com
```

- Within project directory, type `git init` to set up new repository → creates the ‘.git’ sub-dir.
- Add files to staging area using:

```
$ git add <filenames>
```

  NB: Can use wildcards (e.g. `*.txt`) or `-A` to add all files

- Commit staged files to create first project snapshot by typing:

```
$ git commit -m “commit message”
```

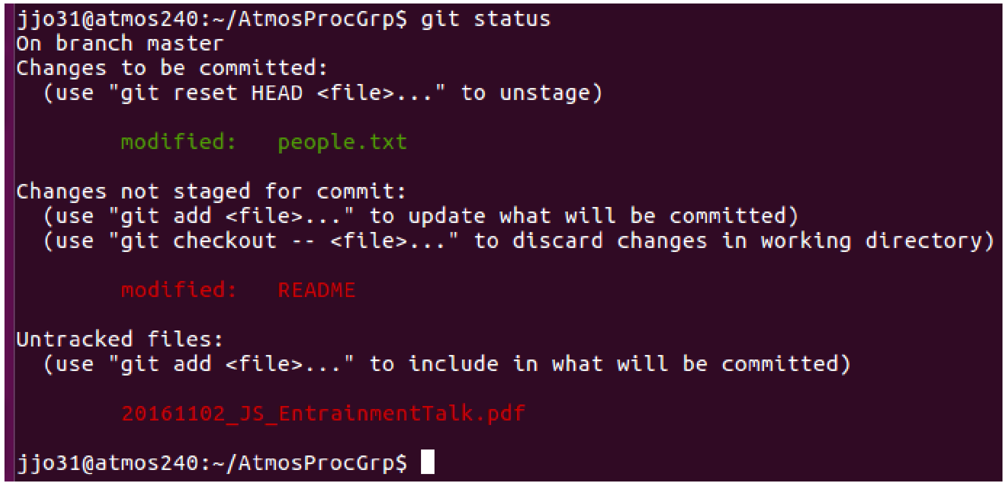
  NB: Each commit requires a commit message. If you just type `git commit`, an in-line text editor is fired up (useful for longer commit msgs)

Status of files

- After initial commit, files can have one of four statuses:
  - Untracked: i.e. new files
  - Unmodified: Not edited since last commit
  - Modified: Edited since last commit but not staged
  - Staged: Edited and then staged ready for next commit
- Very useful command to see status of files:

```
$ git status
```

- Can create a .gitignore file listing files / file patterns not to track/list



```
jjo31@atmos240:~/AtmosProcGrp$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   people.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        20161102_JS_EntrainmentTalk.pdf

jjo31@atmos240:~/AtmosProcGrp$ 
```

File differences and commit history

- The `git diff` command shows the (unstaged) changes to your files since the last commit
- Use `git diff --staged` to see changes to staged files
- Can diff many things, e.g. difference between current file and same file at a previous commit. Use `git diff --help` to see full list of options.
- Use `git log` to see commit history. Again, `git log --help` shows the (many) available options.
- Use `git show <commit-id>` to output metadata and content changes of specified commit

Undoing things

- Use `git rm <filenames>` to remove files (same as standard `rm` but also tells git to stop tracking files). Similarly, use `git mv <filenames>` to rename files.
- Use `git reset [<filenames or commit-id>]` to unstage files (but keep changes) or `git reset --hard [<commit-id>]` to undo all changes to files (staged or unstaged) since last/speficied commit (careful! - all changes and history will be lost).
- Use `git checkout -- <filenames>` to revert files back to how they were at the last commit (careful - all changes will be lost!)
- Use `git clean -f` to remove all untracked (and unignored) files from your working directory (careful – these files will be lost!)
- Useful - can use the “dry-run” `-n` option with many commands to check what would be done.

Collaborating: Creating a ‘remote’

- Collaboration requires a remote copy of the repository that is accessible to all
- For a new project, we need to set up a new ‘remote’:
  - Create an account on GitHub (www.github.com)
  - Select ‘New repository’ and fill in form
  - Copy HTTPS link and back on command line, type:

    ```
    $ git remote add <remote_name> <HTTPS_link>
    ```

    NB: The norm is to call the remote_name ‘origin’
- Use `git remote -v` to see list of remotes
- Use `git push [<remote_name>] [<local_branch_name>]` to push the repository files and commit history to the remote, e.g. `git push origin master`. (NB: ‘master’ is the default name given to the main branch in the local repo with the `git init` command).

Collaborating: Existing remotes

- To download an existing git repository, type:

  ```
  $ git clone <HTTPS_link>
  ```

- After a while, others will likely have pushed more commits to the remote. For your local repo to 'know' about these changes, type:

  ```
  $ git fetch [<remote_name>]
  ```

- To actually implement these changes, type:

  ```
  $ git pull
  ```

  NB: `git pull` automatically fetches new data, so only this command is needed if you simply want to implement the changes without first inspecting what they are. However,...

- If attempting to pull a commit from the remote that conflicts with locally committed changes, the conflicting files will have to be 'merged'. git will attempt to do this automatically, but if the same lines of code have been modified, a manual merge will be needed. A merge procedure generates its own associated commit (more on merging later).
- Similarly, a user won't be able to push any new commits to the remote without first being up-to-date with the remote.

Branching

- Branches are typically used to work on a new development / bug-fix without interfering with the main line of code (until it is ready to be merged into it)
- Each commit stores a pointer to its 'parent' commit. A branch is simply a pointer to a particular commit
- The HEAD is the pointer of the current branch
- For a linear workflow, HEAD points at the master branch. Fig (1).
- A new branch is created using:

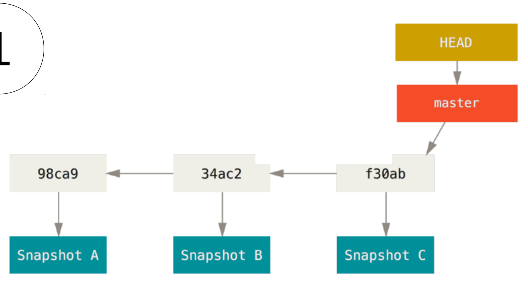
  ```
  $ git branch <branch_name>
  ```

- To switch to a different branch (i.e. move the HEAD), type:

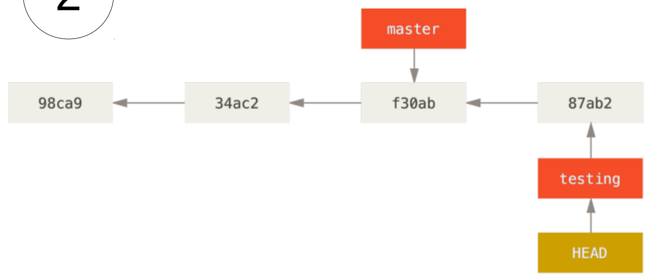
  ```
  $ git checkout <branch_name>
  ```

  NB: `git checkout -b <branch_name>` creates and switches to a new branch in one command. `git branch` on its own lists all branches
- New commits then only move the live branch forward. Fig (2).
- Since checking out the master branch will now reset some files in your working directory, git will only allow this if the changes in your side branch have been committed (or stashed – see later).
- Further commits to the master branch will now result in diverging histories. Fig (3).

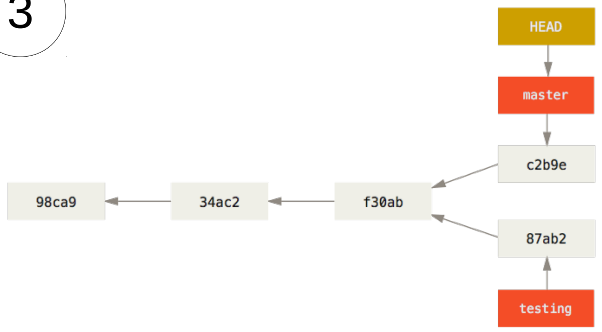
1



2



3

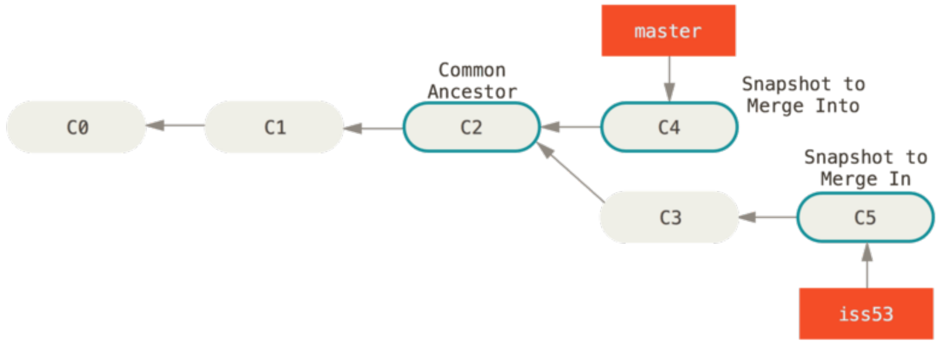


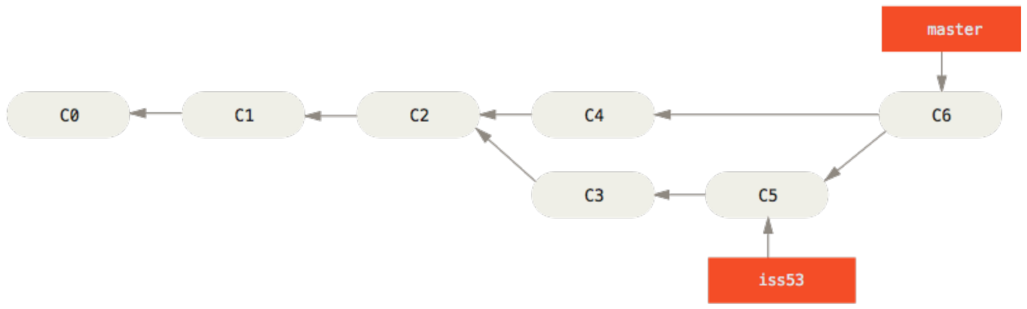
Merging

- Once development / bug-fix is ready on the side branch, it should typically be merged back into the master branch
- To merge, checkout the branch that is to be *merged into*, i.e. master, and type:

```
$ git merge <branch_to_merge_in>
```

- Master branch is often further forward than when side branch was created (due to local commits and/or global commits pulled down from remote) – git must therefore perform a three-way merge using two branch tips and common ancestor
- A new commit is created for the snapshot resulting from this three-way merge, referred to as a “merge commit” → a commit that has more than one parent
- User may have to resolve any merge conflicts manually, then commit to complete merge
- Can then delete side branch using `git branch -d <branch_name>`





Remote branches

- Local branches aren't automatically synced to the remote with `git push` – thus, if you want to collaborate on a branch (or just have a version of it stored elsewhere for safety), you must explicitly push it:

  ```
  $ git push <remote_name> [<local_branch_name>]:<remote_branch_name>
  ```

  e.g. `git push origin Iss39Fix:Iss39Fix_JO`. This allows you to push a local branch to a remote branch with a different name
- Conversely, remote branches are automatically downloaded with `git fetch / git pull` (i.e. no need to explicitly ask for them). Can then use, e.g., `git checkout Iss39Fix_JO` to switch to the pulled-down branch named 'Iss39Fix_JO', `git merge Iss39Fix_JO` to merge it into your current working branch, or `git checkout -b Iss39_JS Iss39Fix_JO` to use it as the starting point for your own new branch called 'Iss39Fix_JS'.
- To delete a remote branch, type:

  ```
  $ git push <remote_name> --delete <remote_branch_name>
  ```

Stashing

- Useful if you're not ready to commit changes in a certain branch but want to move to another branch. Simply type:

```
$ git stash
```

- All modified (staged or unstaged) files will be stored on your 'stack', and the working directory will be 'cleaned', i.e. reverted back to state at last commit, allowing you to switch branches.
- Use `git stash list` to see stored stashes, `git stash apply [stash_name]` to restore the most recent/speficied stash, `git stash drop [stash_name]` to discard a stash, and `git stash pop [stash_name]` to restore the stash and delete it from your stack simultaneously.