Advanced Git

IVS demonstration exercise

Viktor Malík Petr Stodůlka
Red Hat

April 3, 2024

Prerequisites

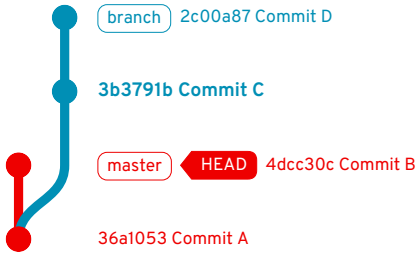
- Basic knowledge of Git commands for:
  - creating commits (`git add`, `git commit`)
  - inspecting current state (`git status`, `git diff`)
  - inspecting history (`git log`, `git show`)
  - working with remotes (`git pull`, `git push`)
  - working with branches (`git checkout`, `git branch`)
  - merging branches (`git merge`, `git rebase`)
- Git commands cheatsheet:
  `https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet`

Git cherry pick

- `git cherry-pick` allows to **copy** a commit from one branch to another

Git cherry pick

- `git cherry-pick` allows to **copy** a commit from one branch to another

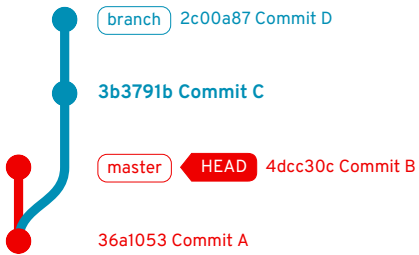



```
git cherry-pick 3b3791b
```


Red Hat

Git cherry pick

- `git cherry-pick` allows to **copy** a commit from one branch to another

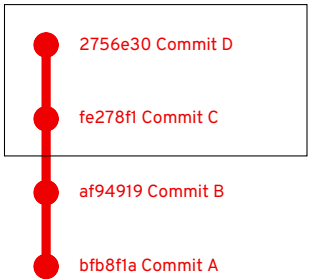



```
git cherry-pick 3b3791b
```

Red Hat

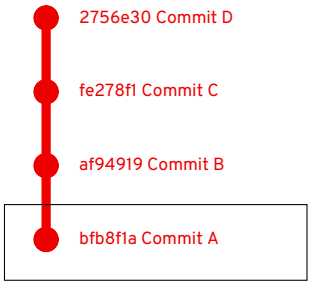
Git commit ranges

- `2756e30..af94919` selects all commits from *Commit D* (inclusive) to *Commit B* (exclusive)



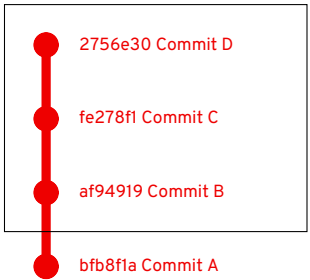
Git commit ranges

- `2756e30..af94919` selects all commits from *Commit D* (inclusive) to *Commit B* (exclusive)
- `af94919^` gives the parent of *Commit B* (*Commit A*)



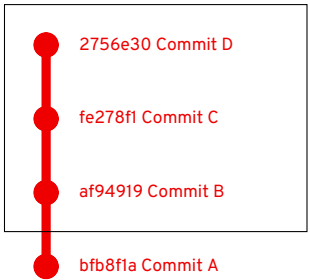
Git commit ranges

- `2756e30..af94919` selects all commits from *Commit D* (inclusive) to *Commit B* (exclusive)
- `af94919^` gives the parent of *Commit B* (*Commit A*)
- Hence, `2756e30..af94919^` selects the commit range including *Commit B*



Git commit ranges

- `2756e30..af94919` selects all commits from *Commit D* (inclusive) to *Commit B* (exclusive)
- `af94919^` gives the parent of *Commit B* (*Commit A*)
- Hence, `2756e30..af94919^` selects the commit range including *Commit B*
- **Note:** the order of references does not matter. `2756e30..af94919^` = `af94919^..2756e30`

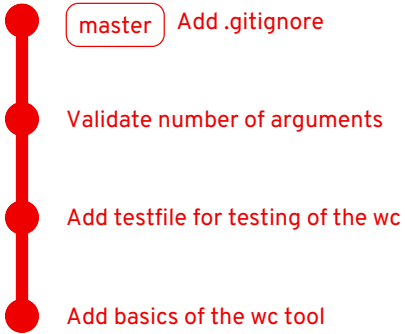


”Advanced” work with Git

Let's start

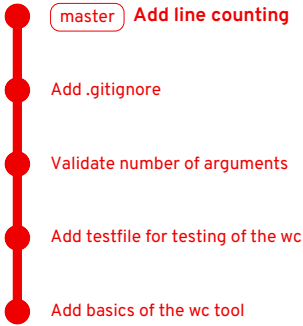
- We'll write a simple tool for counting characters, words, and lines in a file (similar to the `wc` utility)
- We start with a pre-initialized repo containing very basics of the tool: `https://github.com/viktormalik/git-workshop`
- The repo contains:
  - source file `wc.c`
  - testing file `testfile`
  - `Makefile`
  - `.gitignore`

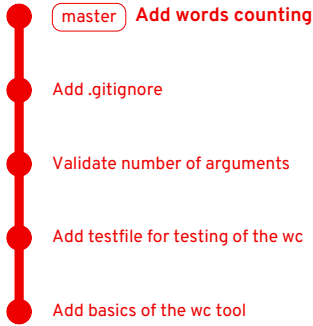
Current status of the repo



Basic team synchronisation

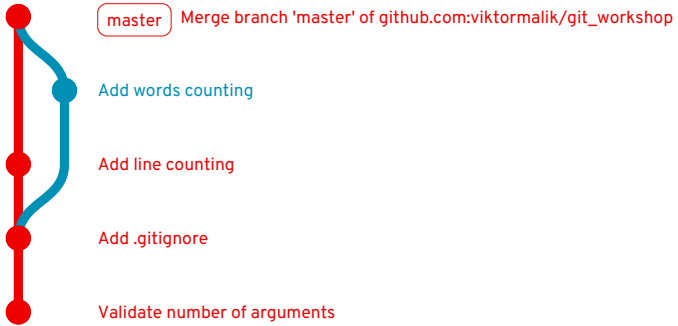
Every member implements a different feature in their *master*





Basic team synchronisation

The second one to push must do a merge (and resolve a merge conflict)



Red Hat

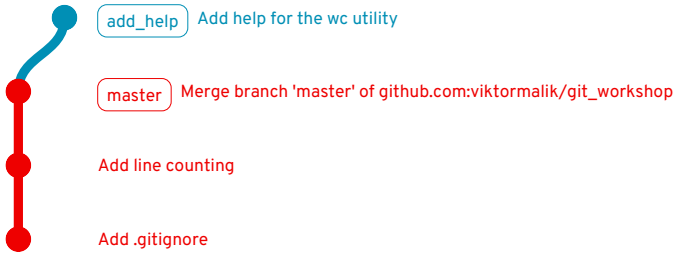
Better team synchronisation

- **This is not a good practice!**
- Always implement new features in **separate branches**.
- Potential merge conflicts should be resolved in the feature branch.
- Ideally, merging into master should be always done using **pull requests**
  - They allow other team members to comment on the changes
  - Changes can be **reviewed** before they get into master
  - Master always contains a working and approved version of the project

Using a feature branch

Let us add help into the tool using a separate branch *add_help*

```
git checkout -b add_help
git commit -m "Add help for the wc utility"
```

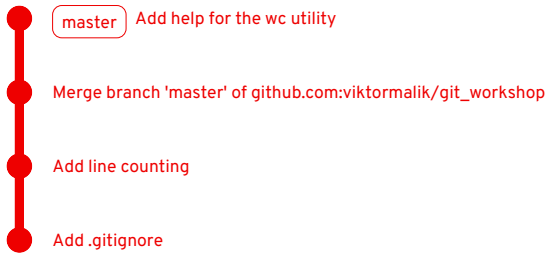


Red Hat

Using a feature branch

Then, we open a **pull request (PR)** from *add_help* to *master*, review it, and merge it using the **“rebase”** strategy.

The state of *master* after the PR is merged:



Moving branches

We start working on a new feature (branch *own-separator*) only to realize that we need to implement something else before. So, we create another branch *option-opt*.

But now, we have two branches pointing to the same commit and we need to **move one backwards**.

Red Hat

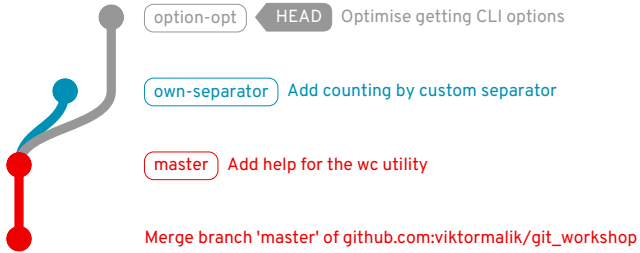
Moving branches

Instead of deleting and re-creating *option-opt*, we can move it **one commit back**:

```
git checkout option-opt
git reset HEAD^
```

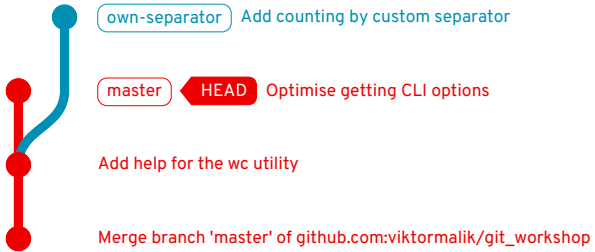
Moving branches

After adding a new commit to *options-opt*:



Moving branches

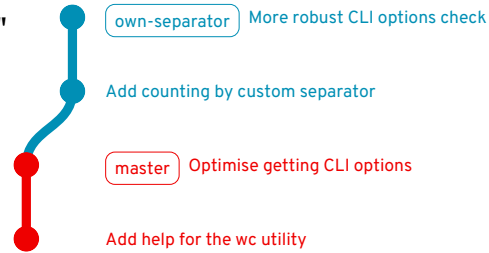
options-opt can be now merged into master while *own-separator* remains a feature branch in development.



Rebasing feature branches

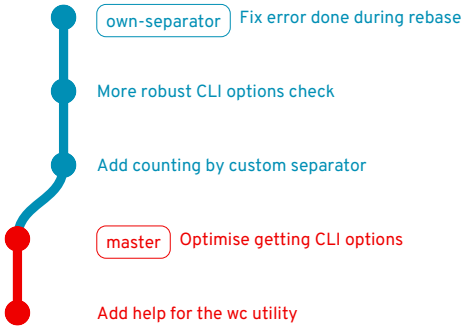
We add more commits to the feature branch and then **rebase** it onto *master* (to avoid creation of a merge commit). This introduces a **merge conflict** which we need to resolve using a **mergetool** (we're using `meld`).

```
git checkout own-separator
git commit -m "More robust ..."
git rebase master
[... merge conflict ...]
git mergetool
```



Rebasing feature branches

We made a mistake during the rebase, which we had to fix with an additional commit.



Red Hat

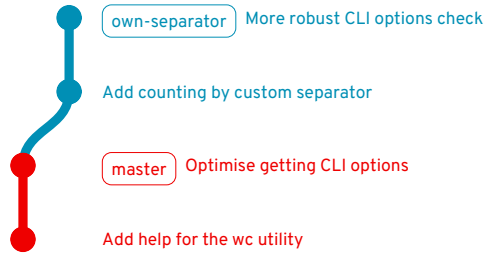
Rebasing feature branches

It is possible to merge the “fix commit” into one of the previous commits using **interactive rebase** (`git rebase -i master`):

Opens up an interactive editor:

```
pick Add counting by custom separator
fixup Fix error done during rebase
pick More robust CLI options check
```

This merges the second (originally last) commit into the first one:



Interactive rebase

- One of the most important Git features in the modern pull request-based workflow.
- Allows to **edit**, **reorder**, **merge (squash)**, or **drop** commits.
- **Rewrites history** – should be only used on feature branches.
- **Never rewrite history of master!**
  - Other developers would not be able to do `git pull`.

How to rewrite commit history

Option 1: edit commits via interactive rebase

Running interactive rebase and selecting `edit` for the relevant commits:

```
pick c853f71 unify whitespaces (replace t by 4 spaces)
pick 4fe8acb extend gitignore: added .test-playground
pick 1b7ccf1 Add just comments into the code
edit e94003b Improve processing of the cmdline parameters
pick b5917e8 cmdline parsing: filename is not positional anymore
pick 43b6520 Check the input file has been opened
```

How to know the right commits? Use `git blame`.

How to rewrite commit history

Option 2: using fixup commits

Commit with the `--fixup` option:

```
$ git log --oneline -3
43b6520 Check the input file has been opened
b5917e8 cmdline parsing: filename is not positional anymore
e94003b Improve processing of the cmdline parameters
$ git commit --fixup e94003b
$ git commit --fixup b5917e8
```

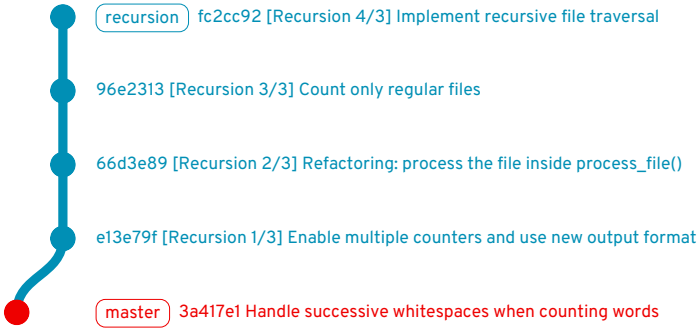
Now, using interactive rebase with `--autosquash` will take care of everything:

```
git rebase master --interactive --autosquash
```

Copying commits from other branches

It is possible to **copy commits** from other branches (e.g. commits implementing useful features from co-workers feature branches) using `git cherry-pick`.

The *recursion* branch:



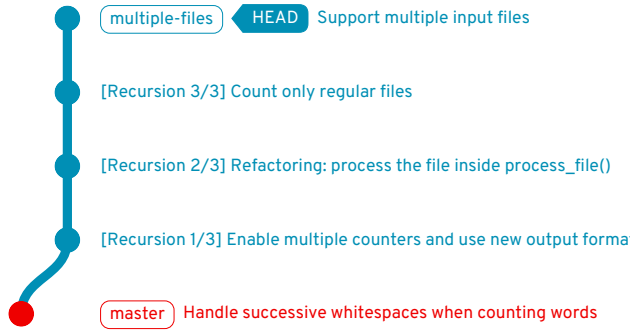
Copying commits from other branches

Now, let's create a new branch *multiple-files*, cherry-pick the first three commits from *recursion*, and add a new commit on top:

```
git checkout -b multiple-files
git cherry-pick e13e79f^..96e2313
git commit -m "Support ..."
```

Equivalent cherry-pick range:

```
recursion@{4}..recursion@{1}
```



Copying commits from other branches

Finally, we rewrite the cherry-picked commits:

```
edit 9abab39 [Recursion 1/3] Enable multiple counters and use new ...
reword 2c403cc [Recursion 2/3] Refactoring: process the file inside ...
reword f85bb09 [Recursion 3/3] Count only regular files
pick Support multiple input files
```

Then, we try to rebase `recursion` on top of `multiple-files`:

```
git checkout recursion
git rebase multiple-files
[... merge conflict during applying [Recursion 1/3] ...]
```

Git tried to apply the first commit from *recursion* (`e13e79f`) but the commit is already in *multiple-files*. Git failed to recognise that since we altered the commit.
The solution is to use **`git rebase --skip`** for such commits.

Hunting bugs in Git history

- We often discover a bug that was certainly introduced **somewhere in the Git history**.
  - There is a revision in the past where certain test works correctly.
  - However, the test does not work now.

Hunting bugs in Git history

- We often discover a bug that was certainly introduced **somewhere in the Git history**.
  - There is a revision in the past where certain test works correctly.
  - However, the test does not work now.
- Git offers `git bisect` that uses **binary search** to localise the commit that caused the bug.
  - `git bisect start` starts bisecting.
  - `git bisect good` marks a commit that does not contain the bug.
  - `git bisect bad` marks a commit contains the bug.
  - `git bisect skip` marks a commit that cannot be evaluated.

Hunting bugs in Git history

- We often discover a bug that was certainly introduced **somewhere in the Git history**.
  - There is a revision in the past where certain test works correctly.
  - However, the test does not work now.
- Git offers `git bisect` that uses **binary search** to localise the commit that caused the bug.
  - `git bisect start` starts bisecting.
  - `git bisect good` marks a commit that does not contain the bug.
  - `git bisect bad` marks a commit contains the bug.
  - `git bisect skip` marks a commit that cannot be evaluated.
- The process can be **automated** using a script that returns 0 on success and a non-zero result on failure.

Git tips and tricks

Cloning repositories with a long history

- If a repo has a long history, it may take long time to clone it.
- If the entire history is no needed, it is possible to use a **shallow copy**:
  `git clone --max-depth N`
- Try it with the Linux kernel:
  `git clone --max-depth 1 https://github.com/torvalds/linux`

Signing commits

- By default, it is not possible to verify that a certain commit was truly created by the person who is stated as the author.
- Theoretically, anyone can set your name and email as theirs and commit on your behalf.

Signing commits

- By default, it is not possible to verify that a certain commit was truly created by the person who is stated as the author.
- Theoretically, anyone can set your name and email as theirs and commit on your behalf.
- To resolve this problem, Git offers **signing commits** using GPG keys.
- GitHub offers a nice tutorial on how to setup commit signing: `https://help.github.com/en/github/authenticating-to-github/signing-commits`

Setup your environment

There are various possibilities on how to ease your life with Git:

- **Git prompt**
  - It is possible to setup Bash prompt such that it shows the current branch, state of the directory, etc.
  - There are many tutorials on how to set the prompt
  - Some alternative shells (e.g. Fish, zsh) include Git prompt by default

Setup your environment

There are various possibilities on how to ease your life with Git:

- **Git prompt**
  - It is possible to setup Bash prompt such that it shows the current branch, state of the directory, etc.
  - There are many tutorials on how to set the prompt
  - Some alternative shells (e.g. Fish, zsh) include Git prompt by default
- **IDE/Editor support**
  - It is useful to see which lines were added/removed/changed from HEAD.
  - Most IDEs and editors offer a way to setup this.

Setup your environment

There are various possibilities on how to ease your life with Git:

- **Git prompt**
  - It is possible to setup Bash prompt such that it shows the current branch, state of the directory, etc.
  - There are many tutorials on how to set the prompt
  - Some alternative shells (e.g. Fish, zsh) include Git prompt by default
- **IDE/Editor support**
  - It is useful to see which lines were added/removed/changed from HEAD.
  - Most IDEs and editors offer a way to setup this.
- **Use tools for history inspection**
  - There is a number of tools for an easier history traversal
  - E.g. **tig**, gitk, ...

Git and IDEs/Editors

Overcome The Doorway Effect of switching to your terminal, examples:

Git and IDEs/Editors

Overcome The Doorway Effect of switching to your terminal, examples:

- **VSCode**
  - Highlight added/changed/removed lines
  - Git blame for each line
  - Commit, push, pull etc.

Git and IDEs/Editors

Overcome The Doorway Effect of switching to your terminal, examples:

- **VSCode**
  - Highlight added/changed/removed lines
  - Git blame for each line
  - Commit, push, pull etc.
- **Vim**
  - **git-gutter**
    - Display line status on the side
  - **vim-fugitive**
    - Full fledged TUI for Git right in your Vim
    - Commit, push, pull etc.
    - `<Esc>:G-cciExample commit<Esc>:x-`

Setup your environment

- **Command aliases**
  - Many Git commands are quite long (or have many options).
  - It is possible to setup short aliases for most commonly used commands.
  - Git offers a way to set aliases:

    ```
    git config --global alias.co checkout
    ...
    ```

    or edit `$HOME/.gitconfig`:

    ```
    [alias]
     co = checkout
     ...
    ```
  - An alternative is to setup aliases via shell

Keep your repo clean

- Delete merged/obsolete branches (locally)
  - `git branch -d` doesn't always work (especially with rebases)
  - `git branch -D` works but be careful not to delete something important
- Same applies for remote branches
  - `git push --delete <remote> <branch>`
  - or enable auto-delete branch on your PRs
  - or use GitHub/GitLab/...web UI
- `git prune` removes unreachable objects (branches, tags, etc.)

Other interesting git commands

- `git difftool` – open mergetool for a specific commit and file
- `git worktree` – checkout a branch into a directory
- `git submodule` – embed another git repository
- `git grep` – grep the entire repository
- `git describe` – find tags related to commit
- `git reflog` – the last recovery option when you break your repo

Useful links

- Atlassian Advanced Git Tutorials
  `https://www.atlassian.com/git/tutorials/advanced-overview`
- GitHub Guides
  `https://guides.github.com`
- GitHub Help
  `https://help.github.com/en/github`

TL;DR

What you should take out of this talk:

- Learn and practice **interactive rebase**
- **Read what Git tells you**, there are often good hints (e.g. for undoing things)
- Keep *master* in good shape

Thank you for the attention!

Your feedback is welcome!

`https://forms.gle/rtxQATGqyct9rNM77`