TRIUMF Science Week - Git Crash Course

Dan Thomson

July 31, 2025

Git Workshop

For a local copy of these slides, you can clone the repository with git:

```
git clone https://gitlab.triumf.ca/Thomson/git-workshop.git
```

The slides are all included in the repository README.md file. The slideshow is buildable with pandoc:

```
pandoc -t slidy -s README.md -o slides.html
```

Once the slides have been generated, you can point your browser to that file on your computer and follow the slideshow.

About Me

- Over 20 years' experience working as sysadmin, DevOps and software developer.
- Currently working in IS&T managing compute and on-site IT infrastructure.
- Most of my work includes IaC tools and a GitOps approach to server management.

Where to Get More Help

- The official git documentation at https://git-scm.com/doc is very well put-together.
- Every git command has a clear and well written git man page associated with it, which is also available in the reference section on git-scm.com/doc.

What is Version Control?

- Version control is a way of tracking changes to a set of files over its development history.
- It often aims to also make collaborative easier and safer by managing contributions to a single repository from multiple people.

Types of Version Control Tools

- Individual, File-Based
- Multi-file, Centralized
- Distributed

As we map out the history of version control tools, we can see how they've been designed to address more complex environments.

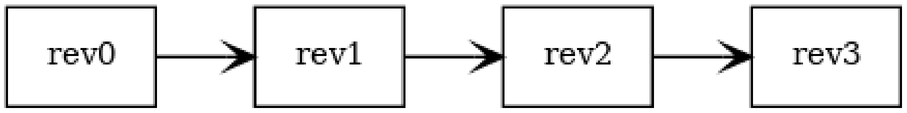
Types of Version Control Tools (cont'd)

Basic Revision Control - RCS

- Relatively simple Version Control System (VCS).
- Manages individual files.
- Each revision is a differential.

1. Check out the file
2. Make changes to the file
3. Check in those changes

example hello.c change history:



Types of Version Control Tools (cont'd)

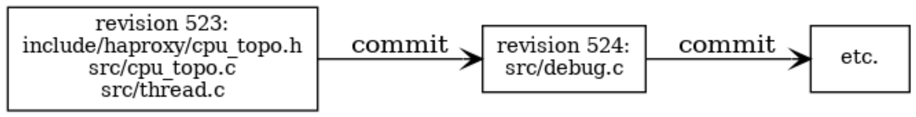
Centralized Version Control - CVS / Subversion / Perforce

- Centralized CVS move the metadata to a central source.
- Enable teams to collaborate.
- Each revision is a differential.

The flow is still:

1. Check out the files (from a centralized repository)
2. Make changes to files in the working copy
3. Check in those changes (ie. check in back to the centralized repository)

example source tree change history with subversion:



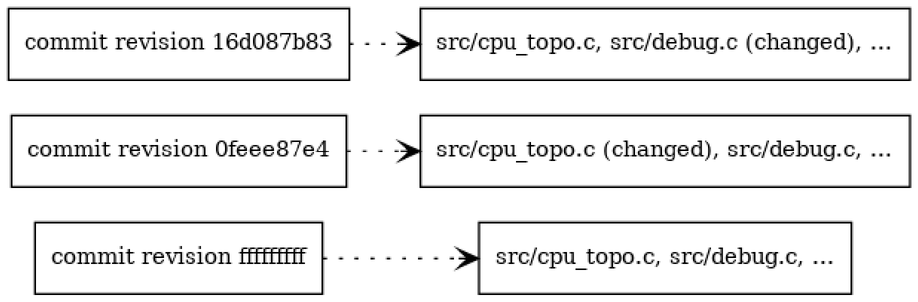
Types of Version Control Tools (cont'd)

Distributed Version Control - Git / Mercurial / BitKeeper

- Each copy is a full repository.
- Can work with multiple upstreams / remote repositories.
- Each revision is a snapshot.

1. Clone the repository
2. Check out your working branch
3. Make changes
4. Commit (ie. check in to your local repository copy)
5. Optionally push your repository metadata to another one

example source tree change history with git:


commit revision 16d087b83
src/cpu_topo.c, src/debug.c (changed), ...
commit revision 0feee87e4
src/cpu_topo.c (changed), src/debug.c, ...
commit revision ffffffff
src/cpu_topo.c, src/debug.c, ...

Getting Started

Git Exercise #0 - Installing Git

Linux

Debian and Variants

```
apt-get install git
```

RedHat and Variants

```
dnf install git
```

Generally, the git package will be simply called `git`, so you can use the package manager relevant to you and install the `git` package.

MacOS You can run:

```
git --version
```

in Terminal. If you don't have it installed, MacOS will prompt you to install it.

You can also use homebrew if you have it installed:

```
brew install git
```

Windows Go to https://git-scm.com/download/win to get git for Windows.

Configuring Git Git will complain if you don't at least have a name and email address configured. If unset, configure your git user and email values:

Eg. for me:

```
git config --global user.name Dan Thomson
git config --global user.email dthomson@triumf.ca
```

Config scopes:

- –global - Set "globally" (in ~/.gitconfig). This applies your config options whenever you use git with your user account.
- –local - Set locally (in .git/config). This applies your git config options to your local repository. These DON'T get pushed elsewhere.
- –worktree - Just like –local, but can be used with worktrees
- –file - Uses a custom file provided by you as the git config file.

Configuring Git (cont'd) There are a lot of possible config options for git. Some I have found useful are:

- `user.signingkey` - Configures a default GPG signing key. This is how you prove that your commits are actually yours, and not somebody impersonating you.
- `credential.helper` - There are various helpers out there that can reduce the toil in continuously entering your password if they're restricted.
- `url.<url>.insteadOf` - You can tell git to replace one URL for another. For me, I use it to convert HTTPs URLs to SSH so that I can always use my SSH key, but you can also just use it as an alias.
- `pull.rebase` - When `true`, git will automatically try to rebase changes on pull, rather than merging them. Overall, this makes for a cleaner change history and should reduce your chances or running into conflicts.
- `merge.tool` - Set your merge conflict resolution tool in cases where you'd like to use "git mergetool" to solve conflicts.
- `diff.tool` - Likewise Set your diff tool in cases where you'd like to use "git difftool" look at code differences. This can usually be the same as your merge.tool.

More info can be found at https://git-scm.com/book/en/v2/Customizing-Git-Git-Configuration.

Configuring Git (cont'd)

- Signing your git commits with a GPG key is the only way to prove that you made the commit.

Create a GPG key:

```
gpg --full-generate-key
```

tell git to use it:

```
git config --global user.signingkey <my key>
```

Once git knows about your key, you can either commit with the "-S" flag:

```
git commit -S -m "My commit"
```

or better yet, tell git to use it all the time:

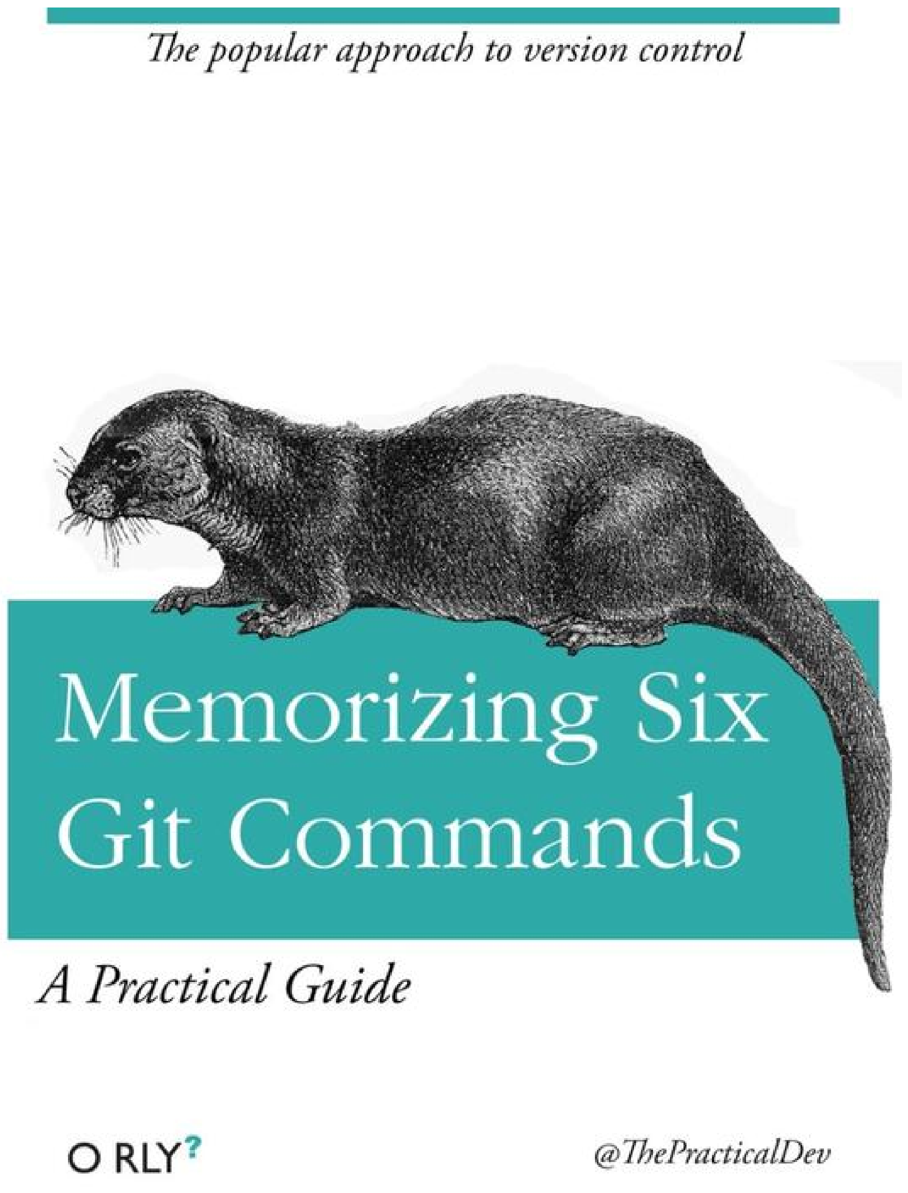
```
# Sign all tags commits
git config --global tag.gpgSign true
```

Github has good documentation on the whole process:

https://docs.github.com/en/authentication/managing-commit-signature-verification/generating-a-new-gpg-key

Git Exercise #1 - Git for Developers

Let's begin looking at Git in a bit more depth, starting with a small example.


The popular approach to version control
Memorizing Six Git Commands
A Practical Guide
O RLY?
@ThePracticalDev

Initializing a Repository

- Start by forking or importing the progit2 repository in gitea.
  - Fork if you're in Github
  - Import/Mirror in Gitea or Gitlab.
- Once forked, clone your copy of the repository:

```
git clone <repo url>
```

The `<repo url>` should look something like `https://gitea.triumf.ca/<username>/progit2.git`.

Once checked out, enter the directory where you've cloned the repository and check the status:

```
git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

Use `git remote` to show the remote source of your repository:

```
git remote -v
origin  https://github.com/progit/progit2.git (fetch)
origin  https://github.com/progit/progit2.git (push)
```

Initializing a new Local Repository

- Repositories can also be created with `git init`.

Create a new directory tree anywhere on your computer.

```
mkdir -p ~/tmp/my-project
cd ~/tmp/my-project
git init
```

Nothing seems different (in bash at least)

```
ls
```

But looking closer shows us that the git metadata directory has been created for us:

```
ls -la
total 20
drwxr-xr-x  3 dthomson dthomson  4096 Jul 29 14:17 .
drwxr-xr-x 22 dthomson dthomson 12288 Jul 29 14:17 ..
drwxr-xr-x  7 dthomson dthomson  4096 Jul 29 14:17 .git
```

Initializing a New Local Repository

- We can turn our local repository into a remote repository:

```
git clone --bare my-project /tmp/my-project.git
git remote add origin /tmp/my-project.git
git remote -v
```

- Or you can copy your current repo somewhere else and use it as a remote:

```
cp -r .git /tmp/my-project.git
cd /tmp/my-project
git config core.bare true
# Show the existing remote repositories managed in this Git repository
git remote -v
# Add our repository under the name 'origin'
git remote add origin /tmp/my-project.git
# Show the updated remote repositories.
git remote -v
origin  /tmp/my-project.git (fetch)
origin  /tmp/my-project.git (push)
```

Since git repositories are self-contained, it's always possible to play with local instances and copies without having to worry about breaking anything, or causing problems. These make the perfect sandbox to work through issues.

Making Edits in the Working Directory Let's try doing some updates.

- Changes should go in a branch! Please make one, even for your personal project.
    - Branching using git locally looks like:

```
git branch my-branch
```

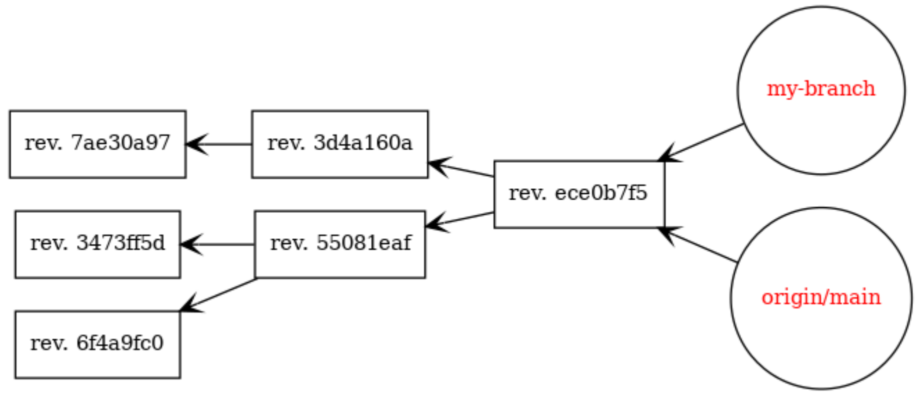


Figure 1: (some) local branches

- Branching can be though through Github/Gitlab/Gitea as well:

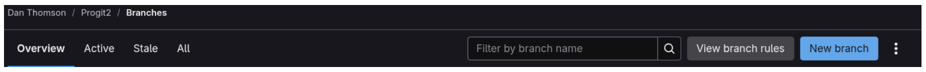


Figure 2: Branching inside Gitlab

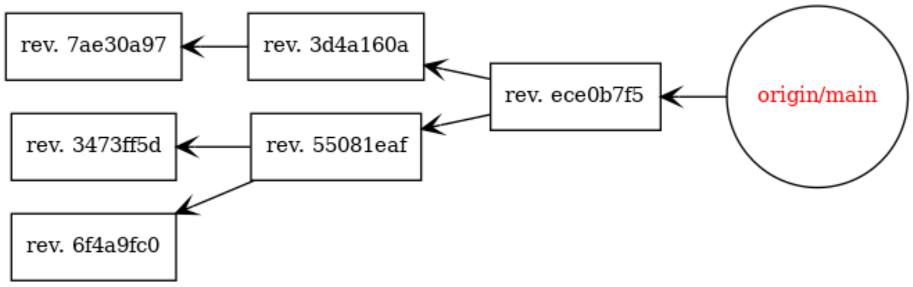


Figure 3: Local branches from remotely branching

Where's my branch?

Staying Up To Date

- Let's inspect things a bit.

The people who have branched locally probably have a different state in their repository compared to people who branched in the web.

What does `git branch` show?

- If I created a branch locally (called `my-branch`)

```
git branch
* main
  my-branch
```

- If I created a branch on the web:

```
git branch
* main
```

Staying Up To Date (cont'd)

- How can you get access to the branch you created remotely?

```
git fetch
From gitlab.triumf.ca:Thomson/progit2
 * [new branch]        my-branch  -> origin/my-branch
```

- Great, now that I've run `git fetch`, I can see the branch now, right?

```
git branch
* main
```

Okay, maybe not. What gives?

Staying Up To Date (cont'd)

- There's a difference between what git metadata I'm holding on to (the stuff in my .git directory) and what I have in my "working tree".

We can see that git knows about the new branch by asking it to check remotes:

```
git branch --remotes
  origin/HEAD -> origin/main
  origin/asciidoctor_2.0.10
  origin/dependabot/bundler/asciidoctor-2.0.23
  origin/dependabot/bundler/asciidoctor-fb2-0.8.0
  origin/dependabot/bundler/asciidoctor-pdf-2.3.19
  origin/dependabot/bundler/epubcheck-ruby-5.2.1.0
  origin/dependabot/bundler/pygments.rb-3.0.0
  origin/filtered_svg
  origin/main
  origin/my-branch
```

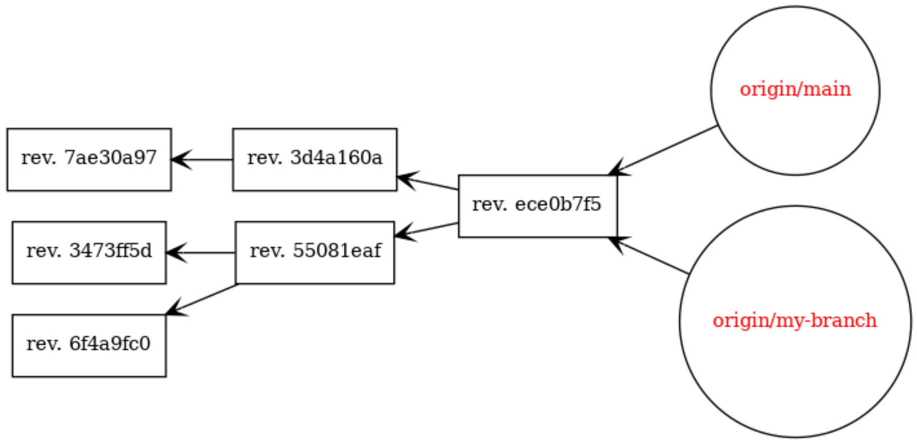


Figure 4: remote graph

Staying Up To Date (cont'd) Since git is now aware of the remote, we can switch to it:

```
git checkout my-branch
Switched to branch 'my-branch'
Your branch is up to date with 'origin/my-branch'.
```

- This obviously applies to much more than branches. It's good to remember that updates from remote repositories, need to be done in 2 steps: fetching changes from remote, then applying those changes locally.

(Finally) Making Edits in the Working Directory

- We should now all be in our own branch. Let's start making some edits.
- All of the files in this repository that end in `.asc` are contents of the git book. You can pick any and start editing them.
- Once your changes have been made, we'll take a look at how to put our changes back into git.

Staging For the sake of simplicity, let's assume I made a change to the `README.asc` file.

- Before doing anything, let's double check the status of our git repo:

```
git status
On branch my-branch
Your branch is up to date with 'origin/my-branch'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.asc
```

Git shows that it knows we made a change, but isn't prepared to do anything yet.

Staging (cont'd)

- Let's make Git aware of the README.asc file:

```
git add README.asc
```

… and check the status again:

```
git status
On branch my-branch
Your branch is up to date with 'origin/my-branch'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   README.asc
```

The file hasn't yet been added, but git is now tracking the file, up to the latest change. You'll often be editing multiple files per commit, but the procedure is the same:

```
git add file1 file2 file3 ..etc
```

- What happens if you change the file again after staging it?

Undoing Staging

- There are a few ways of un-staging your files.

git restore If you stage a file and run `git status`, you'll probably see a message that looks something like this:

```
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
```

- `git restore` is primarily used to restore the state of a file to some existing revision in the git history.
- Using the `--staged` flag in this instance is a special case, indicating that the version of the file in the `index` snapshot should be affected.
- With `--source` we can pick any revision from which to change the contents of some file.

Undoing Staging (cont'd)

git reset

- More of an advanced feature.
- `git reset` is the opposite of `git add` in most cases.
- Unlike `git restore`, this acts on the metadata stored in `.git`, and NOT the working copy file.
- Can be very helpful, but also very dangerous, in the context of undoing a file staging, it should be safe and fairly easy to `git reset <myfile>`.

git reset (cont'd)

- With certain flags (`--soft`, `--hard`) reset will move the `HEAD` of your branch to the selected target.
- My opinion: unless you're managing a git repository (ie. you have merge and/or direct push access to the default branch) this isn't something you should do; work in a branch and squash your commits to keep your history clean.

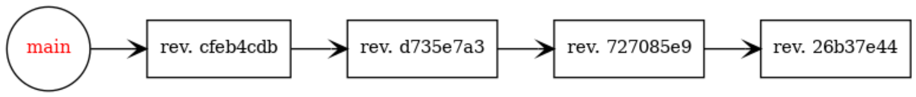


Figure 5: sample git history before reset

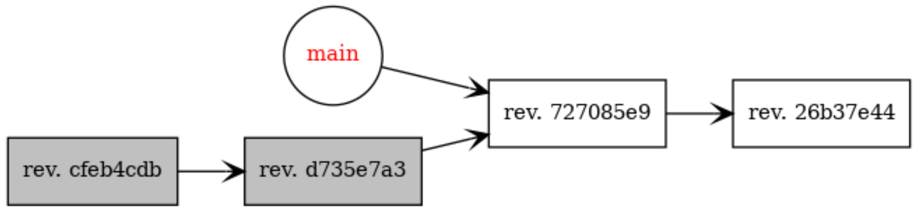


Figure 6: sample git history after reset

Committing When you're happy with your changes, you can commit your changes:

```
git commit
[my-branch d3a51e8b] I made the readme better
 1 file changed, 2 insertions(+)
```

or

```
git commit -m "I made the README better"
[my-branch d3a51e8b] I made the readme better
 1 file changed, 2 insertions(+)
```

What does git think about the repository now?

```
git status
On branch my-branch
Your branch is ahead of 'origin/my-branch' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

- You can update your *last* commit easily enough with the `--amend` flag:

```
git commit --amend -m "This is a better message"
```

Committing (cont'd)

- Try your best to keep good commit hygiene. Every commit should address a single issue. If you fixed two small bugs in the same file, that's two commits; if you fixed a spelling mistake, cleaned up whitespace and fixed a preprocessor condition, that's three separate commits.
- git commit message should also MEAN SOMETHING! You should be able look at your commit messages and have some idea of what you were doing at the time.

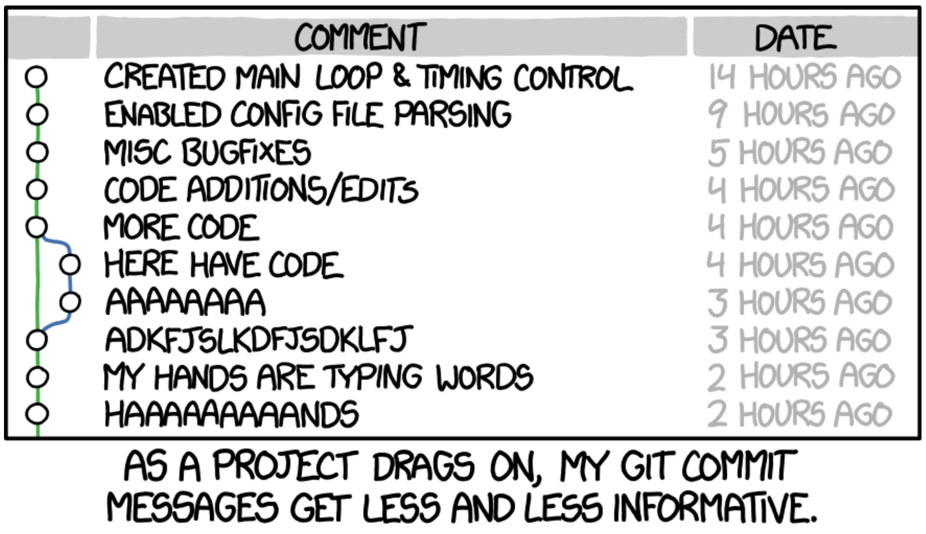


Figure 7: Requisite xkcd comic

Reverting

- `git revert` is how we undo previous changes by adding “inverse” commits to our history.
- Commits are applied automatically.
- Multiple commits can be referenced at once.

If we didn’t like our last commit, we can use `git revert` to undo it:

```
git show HEAD
commit d3a51e8b434f5d2714650a26f095d72426eb2f46 (HEAD -> my-branch)
Author: Dan Thomson <dthomson@triumf.ca>
Date:   Tue Jul 29 16:08:11 2025 -0700

    I made the readme better
# ...

git revert HEAD
[my-branch 496b267b] Revert "I made the readme better"
 1 file changed, 2 deletions(-)
git show HEAD
commit 496b267b3396a8575e842455c600db7604ba866c (HEAD -> my-branch)
Author: Dan Thomson <dthomson@triumf.ca>
Date:   Tue Jul 29 16:13:10 2025 -0700

    Revert "I made the readme better"

    This reverts commit d3a51e8b434f5d2714650a26f095d72426eb2f46
# ...
```

Reverting (cont'd)

- We can further see the results of our initial commit and then our reversion.

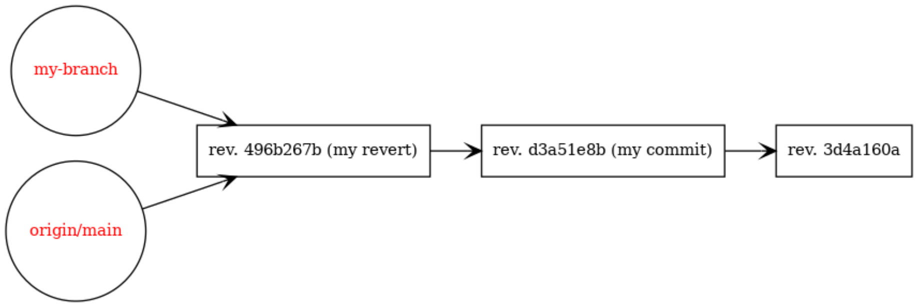
```
git log HEAD~~..HEAD
commit 496b267b3396a8575e842455c600db7604ba866c (HEAD -> my-branch)
Author: Dan Thomson <dthomson@triumf.ca>
Date:   Tue Jul 29 16:13:10 2025 -0700

    Revert "I made the readme better"

    This reverts commit d3a51e8b434f5d2714650a26f095d72426eb2f46.

commit d3a51e8b434f5d2714650a26f095d72426eb2f46
Author: Dan Thomson <dthomson@triumf.ca>
Date:   Tue Jul 29 16:08:11 2025 -0700

    I made the readme better
```



Inspecting Changes

- `git diff` is an indispensable way of seeing what's changed.
- Must involve at least 2 revisions for comparison, sometimes the "other" revision is implied.
- A great way to double-check your changes before committing, including checking your staging area.

Eg. from our sample git repository, I can see what's changed between tags `2.1.446` and `2.1.447`:

```
git diff 2.1.446..2.1.447
diff --git a/book/09-git-and-other-scms/sections/client-svn.asc b/book/09-git-and-other-scms
index 502e4998..de330a1b 100644
--- a/book/09-git-and-other-scms/sections/client-svn.asc
+++ b/book/09-git-and-other-scms/sections/client-svn.asc
@@ -158,7 +158,7 @@ c3dcbe8488c6240392e8a5d7553bbffcb0f94ef0 refs/remotes/origin/master
 6dcb09b5b57875f334f61aebed695e2e4193db5e refs/tags/v1.0.0
 ----

-Git fetches the tags directly into `refs/tags`, rather than treating them remote branches.
+Git fetches the tags directly into `refs/tags`, rather than treating them as remote branche

 ===== Committing Back to Subversion
```

Git Topics

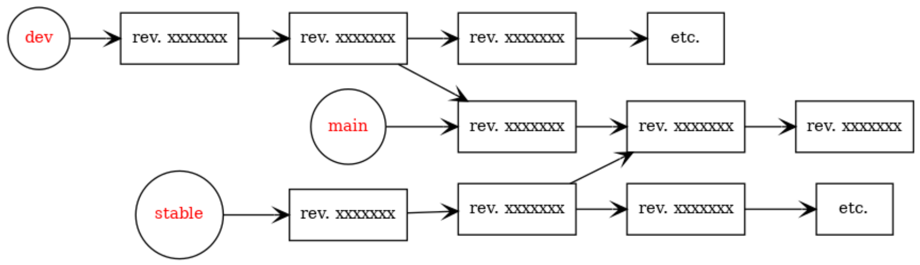
Branches and Branch Workflows

- Branches trace different commit histories from the branch head, following parent commits all the way back.
- Since all commits are snapshots, branches are computationally cheap, and are therefore easy to create and destroy.
- Most branches will be created to isolate work with the intent of eventually re-integrating those changes back into another branch through a `merge`.

Workflow Types

- Branching and integrating branch changes have led to some different common workflows.

Long Running Branches This approach uses one or a few long-running branches that are perpetually updated with merges from other sources.



- Periodically, these branches will be merged "up" into more stable branches as they become ready. In the above example, you may push "dev" -> "stable" when it gets properly tested and cleaned up, and "stable" can be merged into "main" when it's ready for release.

Topic Branches

- A "topic" can be a feature, bugfix, documentation update, etc.
- Topic branches can apply to changes that only require a single commit.

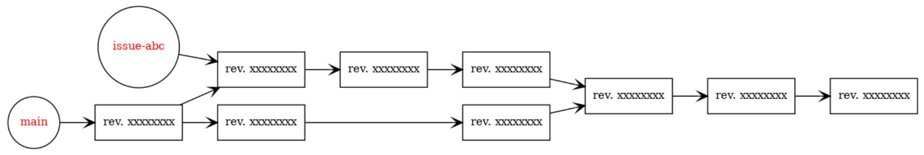


Figure 8: topic branch history

“Standard” Git Flow

- Most projects use topic branches for pretty much everything.
- Branches all come from the default branch, and all re-integrate with there once work has been completed, tested and reviewed.
- Releases are produced using code that sits in the default branch.
- Integrates well with the distributed nature of git and the main Git hosting sites like Github and Gitlab.

The progit2 is a good example of this. Try showing the logs with graphing enabled:

```
git log --graph --abbrev-commit --date=relative --branches
```

“Standard” Git Flow (cont’d)

```
*   commit ece0b7f5 (HEAD -> main, tag: 2.1.448, origin/test, origin/my-branch, origin/main,
|\  Merge: 55081eaf 3d4a160a
| | Author: Ben Straub <ben@straub.cc>
| | Date:   5 days ago
| |
| |     Merge pull request #1975 from Sherry520/index_A
| |
| |     Add index for files in Appendix A
| |
| * commit 3d4a160a
| | Author: Yuhang Guo <22561797+Sherry520@users.noreply.github.com>
| | Date:   12 months ago
| |
| |     Add index for file in Appendix A
| |
* |   commit 55081eaf (tag: 2.1.447)
|\ \  Merge: 6f4a9fc0 3473ff5d
| | | Author: Ben Straub <ben@straub.cc>
| | | Date:   4 months ago
| | |
| | |     Merge pull request #2029 from ablomm/client-svn-grammar-fix
| | |
| | |     Fix grammar on how git svn fetches tags
| | |
| * | commit 3473ff5d
|/ /  Author: Andrew Blommestyn <71096624+ablomm@users.noreply.github.com>
| |   Date:   4 months ago
| |
| |       Fix grammar on how git svn fetches tags
| |
| |       The wording "rather than treating them remote branches." is missing a preposition
| |
* |   commit 6f4a9fc0 (tag: 2.1.446)
```

Workflow Types

- If you're contributing to a project as a developer, the flow will probably already be decided for you. It's your job to learn how the project maintainer is managing contributions.
- If you're maintaining a project, start with branching for managing changes early. At TRIUMF we have a lot of projects that quickly go from "I was just playing around with this on the side" to one with many active users and increasing requirements for features and quality.

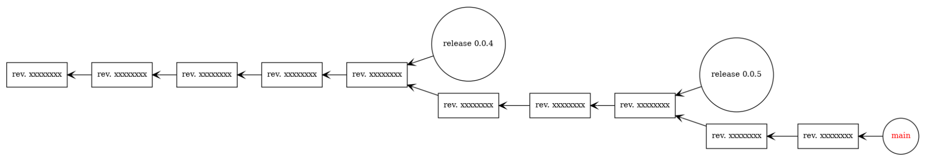
Tags

- Tags are just labels, applied to commits.
- They can be `annotated` (with `-a`) or "lightweight" (with no annotation).
  - Annotated tags include: creation date, the tagger name and e-mail, a tagging message, and an optional GnuPG signature
  - Lightweight tags are just markers for commits.
  - Some git commands skip lightweight tags by default.
- Annotated tags usually denote releases, or some significant milestone in history.
- CI/CD tools can recognize tags and automatically perform some custom action.
- Hosted git providers will apply different permissions rules to tags than they do with branches.

```
git show <my tag>
```

or

```
git diff <my tag>..HEAD
```



Distributed Git

Exercise #2 - Distributing / Remote Sources

- Git was built to make it easy to distribute code and changes to arbitrary external repositories.
- As we've seen, this doesn't have to be a hosted service as long as you can access the repository *somehow* (eg. https, SSH, a different directory)
- Need to bear in mind the differences in state between your local repository and remote repository.

Managing Remote Sources

- Your git repository can have multiple remote sources.
- Each source is named, with the default name being "origin". You should be able to see our gitea instance (or your chosen git provider host) listed in your git remotes with the `git remote --verbose` command.
- A remote can be added with `git remote add`, but must have a unique name.
- A remote URL can be changed with `git remote set-url <name>` and removed with `git remote <name>`.

Eg. adding a new remote repo for my project called "upstream":

```
git remote add upstream https://gitlab.triumf.ca/Thomson/my-project.git
git remote --verbose
```

. . . changing it to use SSH instead of HTTPs:

```
git remote add upstream git@gitlab.triumf.ca:Thomson/my-project.git
git remote --verbose
```

or replacing `origin` with my SSH remote URL instead:

```
git remote set-url upstream git@gitlab.triumf.ca:Thomson/my-project.git
git remote --verbose
```

Distributing / Remote Sources (cont'd)

Fetching From Remote

- `git fetch` will sync the remote repository data up with your own git metadata.

```
git fetch
```

or

```
git fetch <my-upstream>
```

Eg. if a branch was created remotely, you can fetch and then list your branch:

```
git fetch
From gitlab.triumf.ca:Thomson/git-workshop
 * [new branch]      testing    -> origin/testing
git branch -r
  origin/main
  origin/testing
```

- Your "working tree" remains unaffected after a fetch.

Comparing Remote Differences

- Once you've fetched from origin, it's possible to locally inspect differences. Names beginning with `origin/` (or whatever your remote name is) tells git to use the data fetched for that remote source.
- You can compare fully remote revisions, or your local revisions against the remote repository by using your local cache.

Eg. after a fetch, I can check my local HEAD against the remote main branch:

```
git diff origin/main
```

or I can check what's different between 2 remote branches:

```
git diff origin/main origin/my-branch
```

or I can look at the commit history for a remote branch,

```
git log origin/my-branch
```

Merging Remote Changes

- Once fetched, remote changes can be added to your local repository.
- You can merge remote changes into your code base with `git merge`.
- You can also `rebase` the remote repository with `git rebase origin/master`.
- `git pull` is a shorthand for `git fetch` and `git merge` one after the other.
- It's usually preferrable to rebase instead of merge when pulling. You can tell git to do this with by setting config option `pull.rebase` to `true`.

Stashing

- `git stash` can take the changes from your working directory and hide them away.
- Often git will block you from updating from remote due to a dirty local directory. You can:
  - `git stash`
  - `git pull`
  - `git pop`
- You can push and pop multiple times and from arbitrary elements in the stash list (eg. `git stash apply stash@{4}`), or treat it as a stack with `git stash push` and `git stash pop`.
- Messages can be applied with `-m` to indicate what was stashed and you can list the stash contents with `git stash list`.

Git Exercise #3 - Playing with Revision Parameters

- Multiple ways to refer to individual commits.
- Commits are labelled with SHA-1 hashes.
  - Unique Identifiers
  - Checksums
- `HEAD` is a special keyword referring to the latest commit in your current branch. It's generally where you'll be working.
  - Often HEAD is implied. Eg. `git log` and `git log HEAD` refer to the same thing.
- Branches and tags can be identified by their names and disambiguated with a leading `heads/` for branches and `tags/` for tags.

Branch / Tag Names

- Example checking the logs from the `asciidoctor_2.0.10` branch as it is represented on the remote called `origin` (as of my last fetch):

```
git log origin/asciidoctor_2.0.10
commit aaaab2b27fb78050ef52844681d6d59d6d1c4096
Merge: 8897baba bfa64b72
Author: Jean-Noël Avila <jn.avila@free.fr>
Date:   Mon Oct 5 08:48:41 2020 +0200

    Merge pull request #1536 from HonkingGoose/patch-1

    Fix highlighted text in AsciiDoctor 2 migration branch.
# ... etc.
```

There are also tags listed in this repository. If we pick one, we can show the differences between the selected tag and our current HEAD:

```
git diff 2.1.99
```

- References to tags and branches can be disambiguated by a leading `tags/` for tags and `heads/` for branches.

Relative Revision Parameters

- Revisions can be referred to with search parameters and relative-commit parameters, making it easier to traverse commit history.

Selecting Parents

- To see the parent of a revision, you can use the caret character. (In Power Shell, you'll need to do double carets `^^` or quote your string which includes the caret since `^` is an escape character in Power Shell)

Eg. to get the immediate parent of HEAD (`@` is shorthand for HEAD):

```
git show @^
```

- If a number is included after the caret, that denotes the `nth` parent. The default is 1. Normally there's only 1, but merge requests can include multiples (though usually only 2).

Selecting Parents (cont'd) For instance, let's look at the second parent of merge commit `55081eaf`. Git log shows us:

```
git log 55081eaf
commit 55081eaf0bfb4b0f9bab287c0dd035bf604e43bc (tag: 2.1.447)
Merge: 6f4a9fc0 3473ff5d
Author: Ben Straub <ben@straub.cc>
Date:   Thu Apr 10 07:39:49 2025 -0700
```

The second parent of this commit should be `3473ff5d`. This is the commit being merged into the destination branch. We can get information about that commit with:

```
git show 55081eaf^2
```

This is the same as

```
git show 3473ff5d
```

Relative Revision Parameters (cont'd)

- The first parent is often the one we care about, so `^1` is available as a tilde character (`~`).
- The tilde can be entered multiple times to refer to parents of parent.
- The tilde can also be followed by a number that refers to the **`nth`** first parent.

This means that the following commands are all equivalent:

```
git show HEAD~~~~~~~
git show HEAD~7
git show HEAD^1^1^1^1^1^1^1
```

Ranges

- Revision ranges can also be specified to include groups of commits in git's history.
- `^<rev1> <rev2>` - Returns all commits reachable from `<rev2>` but not `<rev1>`. The leading caret in this case can be seen as a type of negation.
- `<rev1>..<rev2>` - A shorthand for the previous notation, since it's so common. I just think of this notation as `"from <rev1> to <rev2>"` but internally, that's not possible.
- `<rev1>...<rev2>` - This is the set of commits reachable from either `<rev1>` or `<rev2>`, but not both. This is called `Symmetric Difference Notation` and is meant to exclude the common ancestor of both revisions.

Using these range formats, you can compare histories with `git log`:

What does this show in the progit2 repository?

```
git log ^761e7e3b ece0b7f5
```

What about this?

```
git log ^ece0b7f5 761e7e3b
```

Git Exercise #4 - Merging Branches

- A “merge” is a way of applying changes from one branch into another.
- There are different strategies to do this, and some are more complicated than others.
  - Cherry-picks
  - Fast-Forwarding
  - Rebasing
  - Merge commits

Cherry-Picking

- Essentially just copying a commit, but that commit can come from anywhere.
- Not a "merge" necessarily, but is used in a lot of merge contexts.

You can test a cherry-pick relatively easily in your own project:

- Add some changes to be cherry-picked (you can also use a change somewhere in the existing history):

```
git switch --create my-new-branch
echo "This is a new file" > my-file.md
git add test.md
git commit -m "Testing Cherry-Picking"
echo "This is yet another file" > my-other-file.md
git add my-other-file.md
git commit -m "This is a commit after my cherry-pick commit"
git checkout main
```

Cherry-Picking (cont'd)

- I'll use log to look into my history to find one of my commits:

```
git log --format=short my-new-branch
commit 519d52d61d187ec53d1a9292c505c09557bdf208 (my-new-branch)
Author: Dan Thomson <dthomson@triumf.ca>

    This is a commit after my cherry-pick commit

commit 4ae06cb70987513a34fa90a21282e7e830679783
Author: Dan Thomson <dthomson@triumf.ca>

    Testing Cherry-Picking

commit a8d1ccd16767c480160fc061b613a79eb8d91318
Author: Dan Thomson <dthomson@triumf.ca>

    Initial commit
```

Cherry-Picking (cont'd)

- I can see that commit `4ae06cb70987513a34fa90a21282e7e830679783` has the commit that I'd like to cherry-pick:

```
# Verifying that we're in "main"
git branch
* main
  my-new-branch
git cherry-pick 4ae06cb70987513a34fa90a21282e7e830679783
[main ed807ad] Testing Cherry-Picking
 Date: Thu Jul 24 11:16:56 2025 -0700
 2 files changed, 3 insertions(+)
 create mode 100644 test.md
```

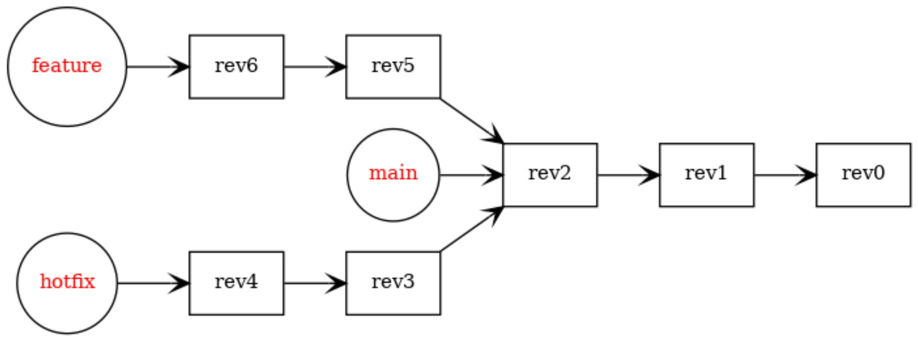
cherry-picking

Merges

Fast-Forwards

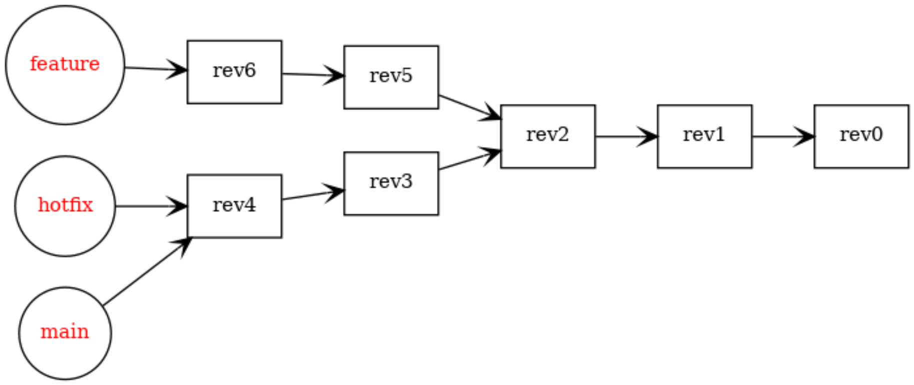
- Fast-Forwards are nice and easy, and will probably be done for you.
- Just moving the index pointer to “catch up” to some other branch.

As an example, what if you find yourself in a situation like this:



Fast-Forwards (cont'd) We want to merge the "hotfix" branch back into the main branch.

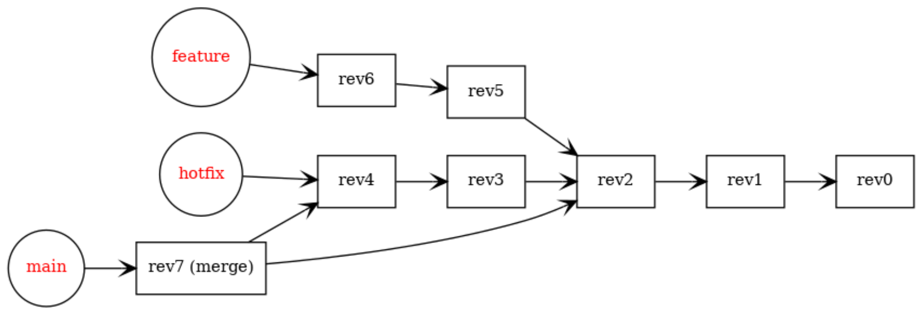
```
git merge hotfix
Fast-forward
 README.md | 2 ++
 1 file changed, 2 insertions(+)
```



Merge Commit

- If we merge with the `--no-ff` flag, we can avoid the fast-forward and force a regular merge commit instead.
- A merge commit combines the two selected revisions with their common ancestor; this is called a "three-way merge".

You might visualize this like so:



- Note that the merge commit has multiple parents. The above example has the same common ancestor as main branch source. This may not be the case in other contexts.

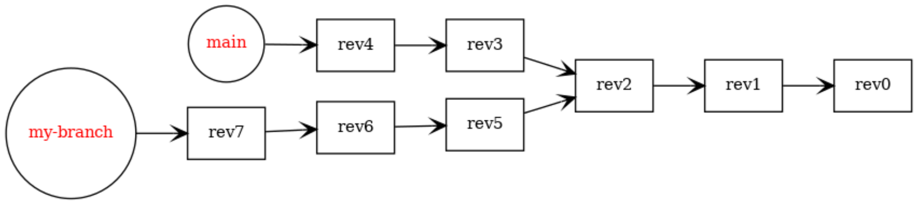
You can try this on your own by creating a branch, committing changes in that branch and then doing a `git merge --no-ff`:

```
git checkout -b noff-test
echo "New Testfile" > new-test.md
git add new-test.md
git commit -m "New Merge Test"
git checkout main
git merge --no-ff noff-test
git log
```

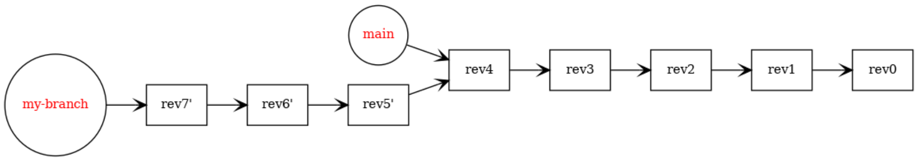
Rebasing

- Rebasing tries to transplant some other branch history in front the destination branch.
- You can imagine the process this way:
  - `git branch temp-branch`
  - `git reset --hard source-branch`
  - `for commit in $(git log --pretty=format:"%h" ^source-branch temp-branch); do git cherry-pick "$commit"; done`
  - `git branch -d temp-branch`
- In the same way that cherry-picks create new commits with the same content, a rebase will generate **new** commits and this may cause issues if you've already pushed your previous changes to a remote repo!

Rebasing (cont'd) Let's say we start with this:



If we rebase `main` into `my-branch`, we should end up with:



Note that a fast-forward situation wouldn't be applicable here, since our branch has commits *after* main.

Rebasing (cont'd) You can try this by creating a new branch, committing a change in main and in the new branch. With both commits done, rebase one branch into the other and check the history.

```
git branch my-rebase-test
echo "## Rebase README section\n\nThis is me adding a commit after branching\n" >> README.as
git add -u
git commit -m "Added a section for my rebase test"
git checkout my-rebase-test
echo "My New Documentation File\n" > doc.asc
git add doc.asc
git commit -m "Rebase Branch Commit"
```

Now what happens when rebasing? You could try rebasing into the branch from main:

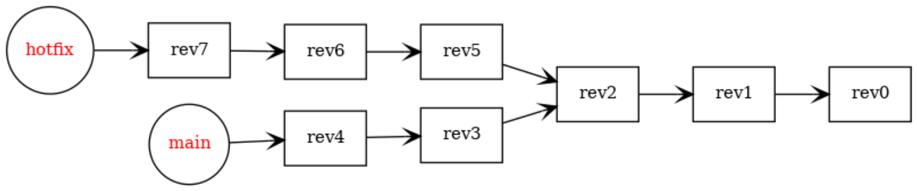
```
git checkout my-rebase-test
git rebase main
git log
```

... or the other way around

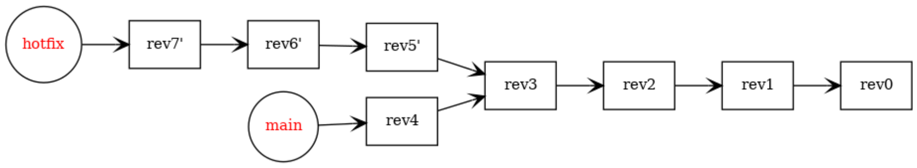
```
git checkout main
git rebase my-rebase-test
git log
```

Rebasing with –onto `git rebase` gives us an option to select the destination revision that we'll rebase "onto". This allows us a little bit more control over where we move our changes.

Eg. we could rebase onto rev3:



We could instead rebase –onto rev3:



Squashing

- `git rebase` also offers the option of interactively rebasing your commit history. This offers the ability to manually select changes, add custom merges and more.
- One of the features in this process is "squashing" commits which refers to the ability to combine multiple commits into a single one.

Why would you want to do this?

- Ideally (IMO), this is a feature that's used sparingly.

`git merge` also provides a `--squash` option. This the more common way that a "squash" gets used.

- Why are merge squashes more common?

Git Exercise #5 - Conflicts

Merge Conflicts

- Changing the same file in multiple branches, and then merging those changes together may result in conflicts.

Eg. a simple C program with a merge conflict.

```
<<<<<<< HEAD:myfile.c
  char mystring[32];
  strncpy(mystring, "somevalue", 32);
=======
  const char* mystring = "somevalue";
>>>>>>> mybranch:myfile.c
```

- The text that appears in your source file tries to help you make decisions on what to keep and what to discard.
- For rebases, the `HEAD` section refers to the tree that you're rebasing on top of. Remember that this is like rolling back and then cherry-picking your changes on top of it, so HEAD is going to refer to the result of the roll back since that's the "target" at that point.
- For regular merges, a `HEAD` is what you'd normally expect. It's the HEAD of your current branch, and usually represents the most recent change in your source tree.
- It's helpful to know which section is which for a merge conflict, but hopefully it's understandable from context as well.
- `git blame` is your friend if you need to identify who made each change and have a talk about how to proceed.

Merge Conflicts (cont'd)

- Let's make a conflict for ourselves to illustrate how this works:

First, in main create `test.c`:

```
#include <stdio.h>
#include <string.h>

int main() {
  char worldstring[32];
  strncpy(worldstring, "World", 32);
  printf("Hello, %s!\n", worldstring);
  return 0;
}
```

commit your changes in main, then create, and switch to the new branch (`git switch -c conflict-test`):

Merge Conflicts (cont'd)

- In the new branch, change the C file to look like:

```
#include <stdio.h>
#include <string.h>

int main() {
  char worldstring[16];
  strncpy(worldstring, "World", 16);
  printf("Hello, %s!\n", worldstring);
  return 0;
}
```

Note that `worldstring` is now only 16 chars long.

- Commit your changes and go back to main (`git checkout main`).

Edit the main C file to look like this:

```
#include <stdio.h>
#include <string.h>

int main() {
  const char *worldstring = "World";
  printf("Hello, %s!\n", worldstring);
  return 0;
}
```

Merge Conflicts (cont'd)

- Now let's merge the changes from the branch into main (my branch is called `conflict-test`):

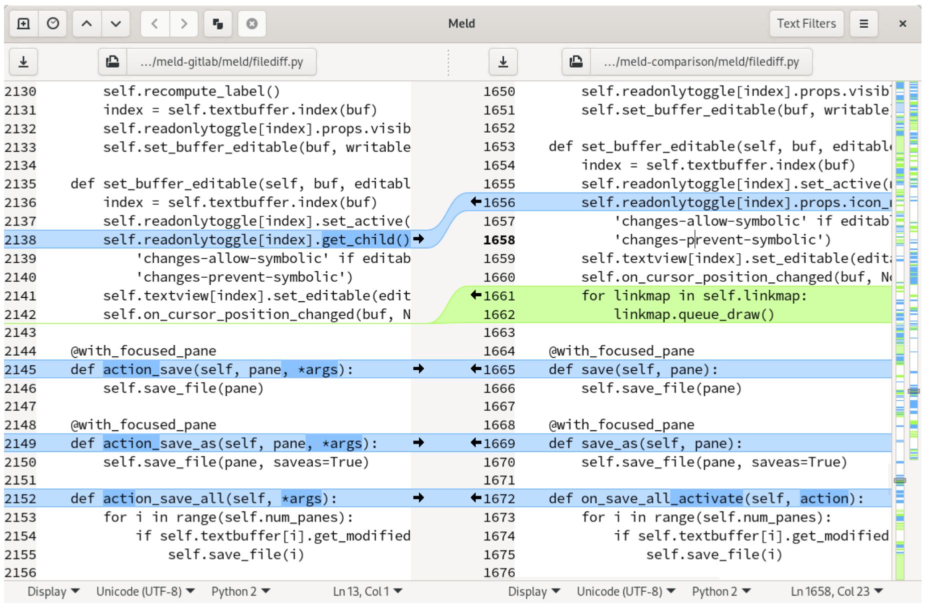
```
git merge conflict-test
Auto-merging test.c
CONFLICT (content): Merge conflict in test.c
Automatic merge failed; fix conflicts and then commit the result.
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   test.c
```

- Let's just fix this by editing the file, or you can use git mergetool (see the next slide) if you'd like.
- When conflicts are resolved, add the file as normal with `git add`.
- When all conflicts are resolved and staged, you can commit.

Merge Conflicts with a Merge Tool

- There are also tools to make conflicts easier to decipher.
- Remember that tools will probably show the 3 sources of a three-way merge: the destination branch, the source branch, and the common ancestor (usually called `base`).
- In Linux, vimdiff, nvimdiff, ediff are options.
- Meld seems to be a popular multi-platform choice for this.



```
        self.recompute_label()
        index = self.textbuffer.index(buf)
        self.readonlytoggle[index].props.visib
        self.set_buffer_editable(buf, writable

    def set_buffer_editable(self, buf, editabl
        index = self.textbuffer.index(buf)
        self.readonlytoggle[index].set_active(
        self.readonlytoggle[index].get_child()
            'changes-allow-symbolic' if editab
            'changes-prevent-symbolic')
        self.textview[index].set_editable(edit
        self.on_cursor_position_changed(buf, N

    @with_focused_pane
    def action_save(self, pane, *args):
        self.save_file(pane)

    @with_focused_pane
    def action_save_as(self, pane, *args):
        self.save_file(pane, saveas=True)

    def action_save_all(self, *args):
        for i in range(self.num_panes):
            if self.textbuffer[i].get_modified
                self.save_file(i)

```

```
        self.readonlytoggle[index].props.visib
        self.set_buffer_editable(buf, writable

    def set_buffer_editable(self, buf, editabl
        index = self.textbuffer.index(buf)
        self.readonlytoggle[index].set_active(
        self.readonlytoggle[index].props.icon_
            'changes-allow-symbolic' if editab
            'changes-prevent-symbolic')
        self.textview[index].set_editable(edit
        self.on_cursor_position_changed(buf, N
        for linkmap in self.linkmap:
            linkmap.queue_draw()

    @with_focused_pane
    def save(self, pane):
        self.save_file(pane)

    @with_focused_pane
    def save_as(self, pane):
        self.save_file(pane, saveas=True)

    def on_save_all_activate(self, action):
        for i in range(self.num_panes):
            if self.textbuffer[i].get_modified
                self.save_file(i)

```

Figure 9: https://meldmerge.org/images/filediff.png

Other Git Resources

Git Exercise #6 - Gitignore

- There are usually files you'd like to keep *out* of your repository.
- .gitignore sits in your directory tree and helps prevent accidental inclusion of files.
- Match filenames, or wildcards:
  - `*` matches every non-slash (ie. non-directory delimeter) character.
  - `?` matches every single non-slash character.
  - `**/` matches match every directory after the slash.
  - `/**` means match everything inside the specified directory.
  - Matches can be negated with an exclamation `!` to whitelist, so blacklisting what you need and then whitelisting specifics is possible, but not normally seen in my experience.
  - Git providers usually offer standard .gitignore files for your project.
- `.gitignore` is a good failsafe for ensuring that you don't commit sensitive details but it's better to just not keep sensitive data in your repo tree at all if you can help it!

Git Exercise #7 - Pre-Commit

- Git hooks can perform custom scripts at certain points during your git life-cycle.
- `pre-commit` is a Python-based tool to install community developed scripts to run at the pre-commit stage in your repository.
- This is a great tool to keep your codebase clean, consistent and free of common errors.
- Install it with:

```
pipx install pre-commit
```

- Once installed, add a `.pre-commit-config.yaml` file to your test repository.

Eg. add YAML syntax checking, end-of-file-fixer (deletes whitespace at the end of a file) and trailing-whitespace (deletes whitespace at the end of a line).

```
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v5.0.0
    hooks:
      - id: check-yaml
      - id: end-of-file-fixer
      - id: trailing-whitespace
```

- Additional hooks exist in this repo, and many more repos exist with ready-made pre-commit scripts.
- Run `pre-commit install` in your repo to enable it. Remember to track the `.pre-commit-config.yaml` file.

Git Exercise #7 - Git LFS

- git is not an appropriate place for large files, especially large binary files.

Why not?

- git LFS is a workaround that manipulates how git tracks certain file types.

Try installing the LFS extension:

```
git lfs install
```

- In your test repository, add an image file.

As an example, you could try the TRIUMF branding page logo:

```
wget https://triumf.ca/wp-content/uploads/2025/02/TRIUMF_Logo_Blue-2048x373.png
```

- Not we'll tell git LFS to track .png files:

```
git lfs track "*.png"
```

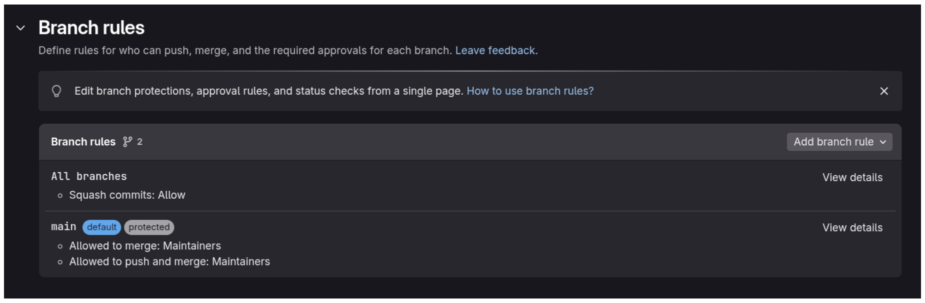
- Let's check the status of the repo after running this command:

```
git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitattributes
```

- `git lfs` creates a file called .gitattributes with some rules on how to manage the tracked git LFS files.
- Commit the .gitattributes file to your repo.
- Add your png/binary file (it should match the `track` fileglob that you configured).
- Your LFS-tracked file should be added somewhere under `.git/lfs/objects`, not the regular `.git/objects` path.
- A small textual link lives in your "real" git repo that won't fill up the history with unwanted data.
- Most hosted services offer some kind of LFS compatibility.

Hosted Git Providers

- Major services are Gitlab and Github. TRIUMF uses Gitlab, but most community software projects use Github.
- These tools are meant for collaboration and offer additional features that can help manage the flow of commits and changes into a repository.
- Branches can be restricted based on who can commit. Usually only a small group of managers have access to the "important" branches like the default or long-running branches in a repo.
- Limits can be placed on tags as well.


Branch rules

Define rules for who can push, merge, and the required approvals for each branch. Leave feedback.

Edit branch protections, approval rules, and status checks from a single page. How to use branch rules?

Branch rules 2 Add branch rule

All branches View details
- Squash commits: Allow

main default protected View details
- Allowed to merge: Maintainers
- Allowed to push and merge: Maintainers

Hosted Git Providers (cont'd)

- Contributing as an outsider - everybody can read your repo, only you can write to it. Changes are submitted through pull requests and/or merge requests to a central repo.
- Often pull requests are meant to be tied to issues, and may have naming schemes for the issue, and/or the branch name. It's good to be familiar with the other repository owner practices when submitting.
- Hosted providers often make convenience files available to you at creation time, and generally they're a good idea. Especially README.md and LICENSE.

Exercise #8 - Create a Pull Request in Gitea

- If there's time, jump into Gitea and either create a new branch, or use some of the changes you've made if you've been following along. Find a colleague's project and try to send a pull request with some changes you've made.
- When a PR has been made, try comparing your own git tree against the other. This might involve updating your own remote repositories to include the source of the pull request.

THANKS!

Thanks for listening to me!

Hope you learned something, and enjoy the rest of Science Week!