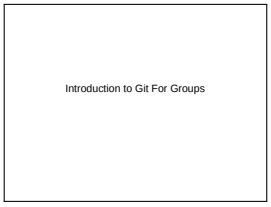
Introduction to Git for Groups: Lecture Notes

Dr Tristan Allwood

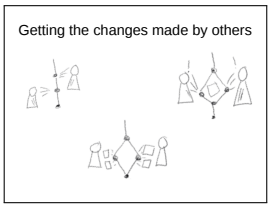
May 2014

Introduction

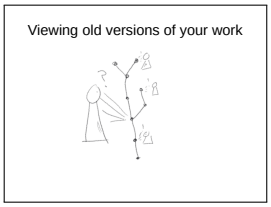


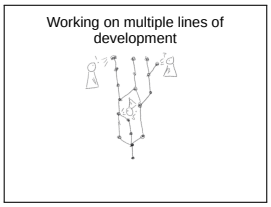
Git is a powerful version control tool.

For the lab exercises so far, you've used it to record changes to a codebase, and to share those changes with others (the lab testing system, your PPT/UTA, etc). However there is a lot more to git. In this lecture, we are going to cover the following:



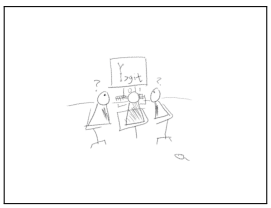
- Getting the changes made by others...
  - when you work at different times
  - when you work at the same time on different files
  - when you work at the same time on the same files
- Viewing / getting back to old versions of your work.
- Working on separate lines of development with your team mates.



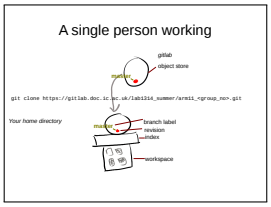


As we work, we will be discussing a model of how git works, and how the commands you run will change that model. We aren't going to talk precisley about how git stores files or creates hashes, the model will be more abstract, but it should be a fairly good intuition for what the commands are doing.

However, all I can do in this lecture is telegraph important commands and ideas that you should be aware of. There is no way you'll suddenly know git from just this lecture, as with programming, the only way to really truly learn it is to use it, to gain experience, to make mistakes and to figure out useful commands and patterns that are helpful to you. Feel free to ask lots of questions during the lectures, in the lab and on piazza!



A single person working



We'll start with the process you've been following in the labs:

For each exercise, a skeleton *repository* has been provided for you on the Gitlab server. This (usually) contains an *object store* consisting of a single *revision*, i.e. a snapshot of some code and files in a skeleton work, and a *branch* called `master`, which points at that revision.

You get a local copy of the repository via:

```
git clone <url>
```

For example, to get your provided group repository for the ARM11 assignment you run:

```
git clone https://gitlab.doc.ic.ac.uk/lab1314_summer/arm11_<group_no>.git
```

This creates a local copy of the repository, and *checks out* the snapshot of the files on the default branch (`master` in this case) into the file system. The files you edit in this snapshot live in the *workspace*, the *object store*, and a temporary working area (called the *index*) are also created and live inside the `.git` directory inside the workspace.

As you progress on your work, it is possible to use git to record snapshots of your work in progress, in-order to send them to other repositories, or to get back to them later. To do this, you *add* a set of changes from the original workspace to the index, and then *commit* a set of these changes to the object store in one go.

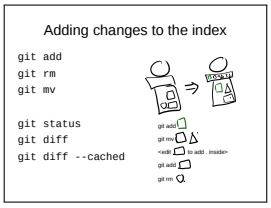
To add changes to the index, you use `git add` on a file that you have created or edited. If the change is to *delete* a file, you can use `git rm`. There is also `git mv` to move a file, but note that git will detect when you move a file based on it's content, even if you forget to use `git mv`.

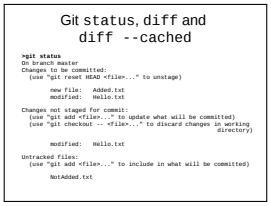
As you are building up a set of changes, you can see what git's view of your repository and index are using `git status`. The changes you've made that are unadded, or uncommitted can be seen with `git diff` and `git diff -cached`.

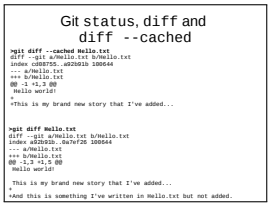
When you have the new snapshot you want ready, you create it using `git commit`. By default, git will open a text editor for you to describe your change in a *commit message*.

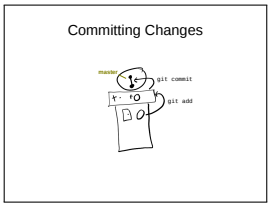
You you create the commit, git will add the change and the new workspace state to the object store. The commit will be associated with a (globally) unique id, which is derived from using a SHA-1 hash all of all the information that made that commit. Git also knows that the active branch is called `master` (by default), and will move that branch pointer along to the new commit.

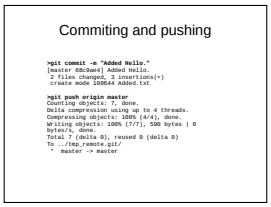
After committing you can send any new commits you have created back to the repository you cloned from using `git push`. Assuming the new location of the `master` branch is a descendant of the old `master` branch on the server, this will update the server's `master` branch pointer, and also the local `origin/master` pointer.

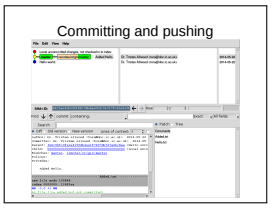


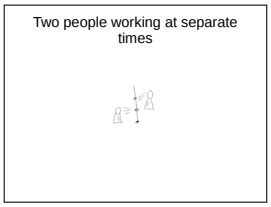












Two people working at separate times

This works fine if you're just working on the lab machines, or on your laptop. However if you're working on both your laptop and home machine, then you have to find a way to get your new revisions to the other machines.

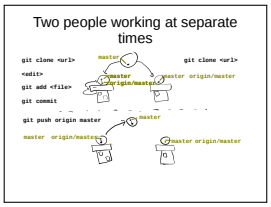
Working with yourself on another machine, or at another time, by the way, is basically the same as working with another person.

Let's try this again, but now two people clone, and then the first makes a change, and adds, commits and pushes it.

The second person now wants to get the latest version of the code. In git this happens in two stages:

- First the changes and revisions have to get into the second per-

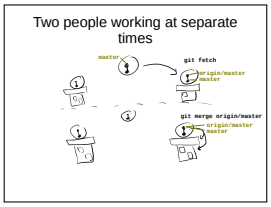
son's object store.



- Then the workspace has to be updated to reflect the state of one of the revisions.

Any changes and revisions can be simply copied across from one repository to another, this can be achieved using:

```
git fetch
```



After running a `git fetch`, it is useful to run `gitk -all` to visualise exactly what you have pulled in, and what the state of your local object store is. Here we can see that the location of master locally is behind that on the server.

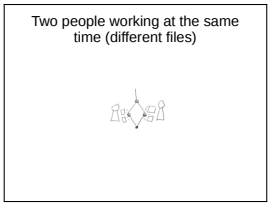
To update the workspace, we can request that our current workspace is merged with the new one.

```
git merge origin/master
```

Git will notice that we are still on the original commit, and that there's a single way to get to the current `origin/master`, so it will apply that change, and update the local `master` branch to point to the same place.

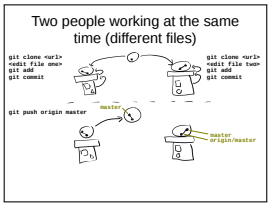
This works the other way - person two could then make a series of changes (using `git add` and `git commit`), and push them back, for person one to `git fetch` and `git merge`.

Two people working at the same time

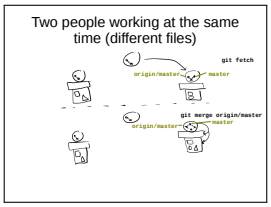


In reality, when two or more people work together, it's likely that they will end up working on the project at the same time.

We'll first consider what happens when they work on different files. For example, Person A creates file A, and Person B creates file B. They both *add* and *commit*.



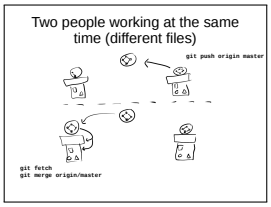
Inside each of their local object stores, the commit will have recorded a new change, and a new revision with the state of the workspace. If Person A pushes (using `git push origin master`), then Person B can get Person A's change and revision into their local object store using `git fetch`. Though the graph of changes has now diverged. Person B has their (committed) change and Person A's change leads to a different state.



However what Person B wants is to see Person A's new file in their workspace. The two changes don't (superficially) conflict in any way (they don't talk about the same files), and so it's fairly simple for git to combine the two changes together to create a new revision.

Person B can now run

```
git merge origin/master
```



and git will combine the committed changes together, building a new revision with both files. git will also move the current branch

(`master`) forward to this combined revision. Person B pushes this change, so Person A can now fetch and merge, and will also see both changes in their workspace.

Remember git won't let Person B push, if the movement of the `master` branch pointer would not be up the tree, but across it. Person B would need to resolve the divergence (i.e. fetch and merge) before git will let them push their change. Later we will see a way of creating new branch labels, which is another way that Person B would be able to push their work.

Of course, people don't always work on separate files. Team members may need to (or accidentally!) edit the same file at the same time - for example while writing different sections of a report, or to bugfix something that's breaking for them in a file someone else is working on. Git is very good at resolving these kinds of conflicts in a way that makes textual sense.

When git detects a conflict during a merge that it cannot resolve, it looks at the two versions of the file you are trying to combine, and it also looks at the common ancestor version to help figure out the change.

It then annotates the conflicting file with information about the different versions, which the user then needs to edit in order to resolve. The annotation will have up to three parts, separated by rows of <, = and > characters.

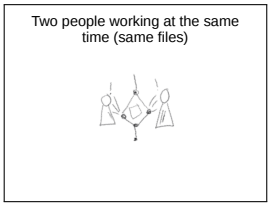
Edit the conflicted file to resolve as appropriate, and then add, and commit the change. Git will prompt with a sensible default commit message set up for you.

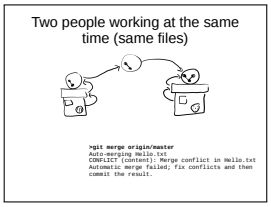
You might want to read up on and experiment with `git mergetool`, as there are many editors (e.g. `vim`, `xxdiff`, `meld`) that can provide a user interface to manage 3-way merges.

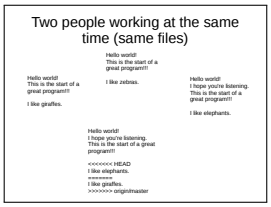
Viewing / browsing old versions of your work

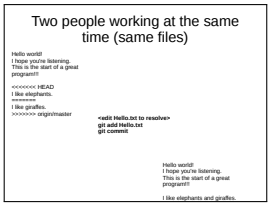
You can use informational tools to visualise the different states and changes stored in the repository. Repository browsers, such as `gitk -all`, `gitg`, or even the web pages on gitlab for a pushed repository are great for exploring who has done what when.

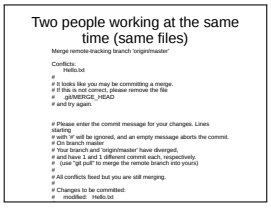
On the command line it is also possible to use `git diff` to see the difference between the current version of a file and previous ones, or `git log` and `git blame` to see what the commits where that affected a particular file or the individual lines of that file. `git show` is also a good way on the command line to get access to a particular snapshot of a file at a particular version.

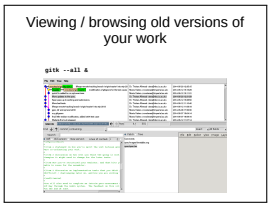


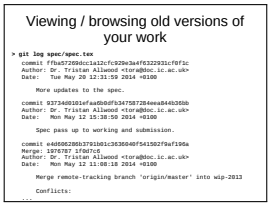


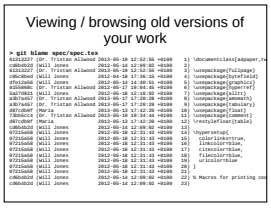












Multiple Branches of Development

As we have seen, git is a very powerful tool for recording, merging and interrogating different versions of work by multiple people. However there is one crucial ability of git's that we have so-far overlooked, which is the ability to easily and cheaply create and manipulate new named branches of work in a repository.

We have seen branches in the revision graph already, when two people create two different commits starting from the same revision. If git believes they are both on the same named branch (e.g. `master`, it won't let both push until the difference is somehow resolved (e.g. through a merge).

However they may be times when you want to share the diverged changes without merging them just yet, or possibly at all. For example, you wish to work on and complete a new feature before merging in the features your team mates are working on, or you may wish to try an experiment or idea out without wishing to lose the place you started from in case it doesn't pan out.

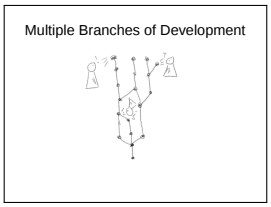
To create a new named branch, you tell git to check you out onto that branch, using `git checkout -b <branch name>`. In your object store this creates a new branch label called `<branch name>` at the commit you're workspace is currently on, and sets your current branch to be that new branch. Now when you commit, the new branch label is moved forward.

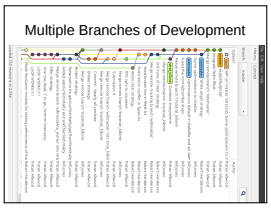
You can explicitly push a particular branch using `git push origin <branch name>`, so others can see it too. This will also create an `origin/<branch name>` branch in the current repository, so git can track when it's local knowledge of a remote branches position changes.

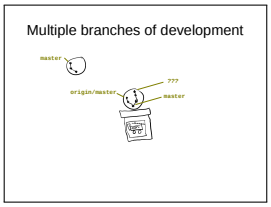
Of course, once you have created several named branches, you want to be able to move your workspace between them. To move the workspace and active branch to a different named branch, use `git checkout <branch name>`.

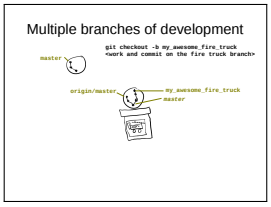
You can create merges between the current active branch and any remote branch using `git merge <branch name>`. This will move the active branch forward, but leave `<branch name>` where it is.

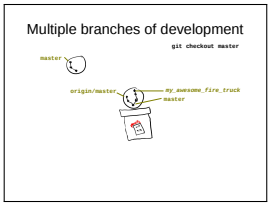
Note that you can actually checkout any particular revision with `git checkout <revision>`. If you do this, git will put you on into a mode where there is no active branch, and will give you a warning message. You can of course create a new branch at that revision using `git checkout -b <new branch>`.

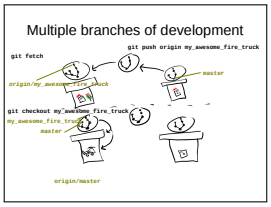


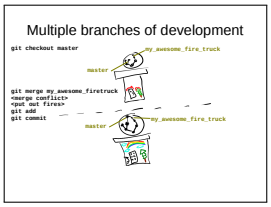


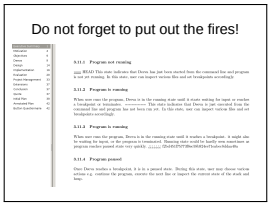












Closing

Please bear in mind this is only scratching the surface of what git can do for you. It has a lot of other very powerful features that are incredibly useful in some cases (e.g. *cherry picking* – taking a single commit from one branch and applying only it on another, *rebasing* – rewriting a branch of history so that it looks like a straight line of development, adding other remotes so you can push or fetch from multiple different repositories, *stashing* – temporary branches managed by git for storing short-lived work-in-progress changes), but they can all be described in terms of the simple underlying model of revisions, commits, branch labels, indexes and workspaces.