- Git is the universal version control software used throughout CS.
For extensive Git documentation, run the command:
  - apropos git
or - apropos git -a tutorial

These commands show the names of a load of different Git tutorials. To access them, simply do man [TUTORIAL]

- Configuring your identification on your local machine:
  - git config --global user.name "Jamie"
  - git config --global user.email "jamiebt@live.co.uk"

These details aren't checked anywhere, they are just listed alongside the changes you make.

- Initialising a Repository on your local machine.
Firstly you make a new folder where you want to keep the project, then you initialise the repo!
  - mkdir new_project
  - cd new_project
  - git init

The git init command will create a .git subfolder. This folder encapsulates all the information & tools necessary for version control & collaboration within the repository.

Now we can start our project. If we create a new file main.c, this won't automatically be tracked by Git. We can see this by running:

- git status

Git status shows the current state of the repository, providing info about:

- Changes in the working directory (which files have been added/modified/deleted since the last commit)
- Changes in the staging area (which files are staged for the next commit)
- Untracked files (files that currently aren't tracked by Git) (main.c)

(files tracked are shown in green & untracked in red)

- To add a file to git, you use

git add [filename]

This command also stages changes to both tracked & untracked files before they are commited. You typically use this command to stage modified, tracked, files. Furthermore, to stage ALL modified/untracked files you simply use the command:

git add .

Once files have been staged they are ready to be commited! To commit then, simply do:

```
git commit -m "commit message"
```

To see a list of previous commits do

```
git log
```

& to see a condensed list of commits (only the commit ID & Message) do:

```
git log --oneline
```

Each commit has information associated with it to make it unque:

- Commit Hash (ID)
  Each commit in git is uniquely identified by a 40-character hexadecimal cryptographic hash.
- Author & Committer info
- Timestamp
- Commit Message
- Parent Commit(s)
  The commit(s) that directly preceed this commit, this allows git to construct a repo history & evolution.

- How to tell git to ignore files.

If there are files you know you don't want tracked, but don't want to be constantly listed as untracked (e.g binary executables), you can tell git to simply ignore them all together.

To do this create a file called `.gitignore`
In this file you can add specific files you want to ignore (e.g the name of a certain executable) or you can ignore a whole filetype altogether using the wildcard symbol.
For example:

```
*.txt
```

this will ignore all txt files in the repo.

- Since each commit is effectively a save point in your code it is good practice to do frequent, small commits rather than a few big ones, so that if a bug is discovered in post it can be rolled back with little damage.

To revert to a previous vesion, first look at the `git log` to see which version you want to rollback to (recall each version has a unique ID) then run:

```
git checkout [HASH]
```

This sets the HEAD pointer (hashcode of latest commit) to the selected commit.

- If you want to undo a commit altogether there are 2 options:
- `git revert [HASH]`
  This adds a new commit that returns the files to the state they were in prior to the specified commit. This is quite safe as you're just adding a new commit rather than deleting anything.
- `git reset [HASH]`
  This undoes a commit by moving HEAD pointer back to the commit with the given hash but leaves the working copy alone.

Git Forges.

The main git forge is GitHub. There is also GitLab & BitBucket.

GitHub communicates with users over SSH (generally), which we saw last week, meaning we already have a key. But for when our key expires:

- Create a new key using ssh-keygen (tutorial last week)
- Go into my GitHub settings & go to SSH/GPG tab
- Paste new public key in there (this is effectively GitHub's known_hosts file)

You should have a seperate SSH key per device to enhance security

- Through GitHub you can create new repos & access their SSH location so you can clone them.

To location will look something like git@github.com:USERNAME/REPONAME.git
& to clone simply do

```
git clone [LOCATION]
```

This clones the repo into a subfolder with the name of the repo.

NOTE, you can also do this via HTTPS, but this requires you to manually enter your GitHub username & password whenever you push changes or interact with a private repo.

- Once you have the repository locally you can see its details by cd'ing in & running

```
git remote show origin
```

In git the word 'remote' is a common term used to refer to a repository that is hosted on a remote server (GitHub)

& origin is a conventional name used to refer to the remote repository from which the local repository was initially cloned

git status in a repo you have cloned from GitHub will show we are up to date with 'origin/main' now rather than just main (if that's the case)

- This command has 4 outcomes.
  1. Up to date - NO commits on local or remote since last synchronisation.
  2. Ahead of remote - We have local commits yet to be pushed to remote
  3. Behind remote - Commits have been pushed to remote that you don't have.
  4. Diverged from remote - Local version & the remote have had different commits since last sync.

To see if there have been any updates to the remote, run:
`git fetch`

This retrieves changes from a remote repo - it doesn't merge them to your local

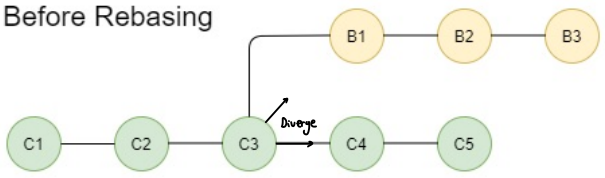
Whereas `git pull` downloads changes to remote and merges them into local

- Resolving Conflicts

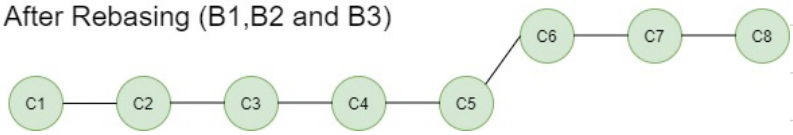
When multiple people are working on the same repo, it is very easy for the code base to diverge.

Resolution Method 1 - Rebasing:

This takes your commits & effectively 'replays' them on top of main branches latest commit. Let's look at this diagramatically.



C4 & C5 are pushes to main that happened while we were creating commits B1, B2 & B3. This is how the divergance formed.



As you can see the rebase will attempt to append the divergant commits to the latest instance of main. (B1-3 → C6-8)
(if succesful)
This will shift the user to ahead of main meaning they can push.

- There is a general consensus that you should only use the rebase method when the divergance was caused by people editing different files. If there are conflicts in the same file you should use `git merge`. The benefit of using this method over merging is that it makes for a cleaner commit graph

Usage: `git rebase origin/main`

- Push Workflow

You should NEVER just blindly run `git push`. There is a procedure you must follow to ensure you don't overwrite other people's code

1. `git fetch`
2. `git status`
3. if ahead of origin/main - `git push`

Check to see you haven't diverged!!

Resolution Method 2 - Merging.

This is more common than rebasing. This method combines changes from two or more branches into a single branch.

Method:

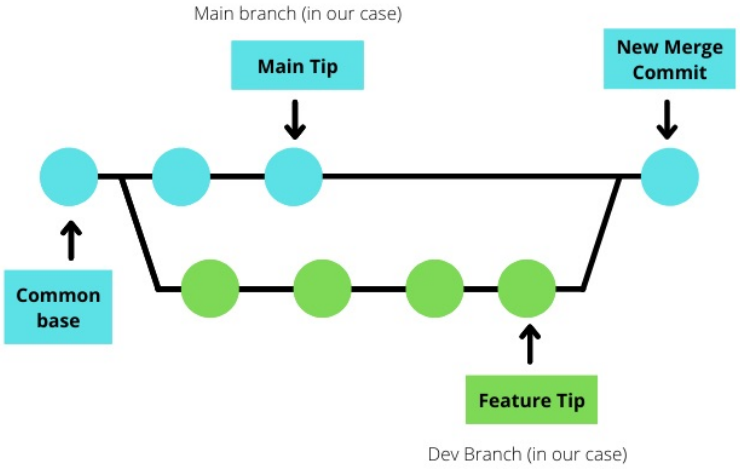
① Commit your changes (ready to push) — git add . / git commit -m "..."
② Fetch changes from origin — git fetch
③ If we have diverged: — git pull (This triggers the merge & is different to git merge)

If there are no conflicts within the same file the merge will be succesful & this will generate a new commit containing the output of the merge. This puts you ahead of main meaning you're ready to push.

(merge conflict)

If there are modifications to the same file, the merge will fail & you have to fix the conflicts manually & then push. Once all conflicts are resolved you simply add, commit & do the push workflow as usual.

Diagramatically:



- Creating a new branch

To create a new branch locally, simply run:

git checkout -b <NAME>

The -b flag creates a new branch & checks it out. In previous versions you had to do:

git branch <name>
git checkout <name>

You should make a commit on this branch. Furthermore you need to set up a tracking relationship between the new local branch (<NAME>) the remote branch (origin/<NAME>) This is done by:

git push --set-upstream origin <NAME>

origin specifies the name of the remote repo
<NAME> specifies the name of the remote branch to push to

You can see all branches on a repository by running:

git branch -a

- Merging development branches into main

Once you have finished developing your feature branch & commited all your changes

① Fetch the latest changes using git fetch
② Checkout the branch you want to merge into - in this case main git checkout main
③ Run git status to make sure you're not behind, if so do git pull.
④ Run git merge feature. This brings feature into main
⑤ Resolve conflicts, test the code & commit merged branch
⑥ Push workflow.

Pull Requests

This isn't a feature inherit to Git, rather a forge tool. They let a team discuss & review a commit before merging it into the shared branch.

Firstly, this is a merge procedure except the merge gets verified & discussed before it is completed.

Procedure:

① Develop your feature branch
② On github.com/REPONAME, select Pull Requests → New Pull Request
③ Set base branch as the one you want to merge into
④ Set compare branch as the one with your changes
⑤ Add a description & create the request.

Once a PR has been submitted a reviewer can add comments highlighting bugs & general feedback, the developer can address these comments by pushing a new commit to the feature branch which is automatically added to the PR discussion. When the reviewer is happy, they approve the request & the merge happens.

- Potential Issue

If there is a modification to the main branch while developing your feature branch, it'll be lost when you attempt to merge. The solution here is to repose your feature branch from the HEAD of main.
git rebase main & fix any conflict issues on your private branch

This might trigger an error when you push because [your ? or origin repository still has the old version] in which case you can completely overwrite it with
git push --force origin BRANCHNAME
Safety Rules:
- Only rebase on private branches
- Only force push on private branches & only if absolutely necessary

- Final Notes on Force Pushing:

If you ever have to rebase or force push on a shared branch - warn everyone so they do not commit to the branch while you're working on it. Basically lock it down while it's happening.

Most organisations will have a main and a develop branch. Develop will have all the feature ~~dev~~-branches + PRs with potential rebasing + force pushes. However main will only recieve Merges from develop once it has been perfected, ensuring a situation where you need to rebase doesn't happen.

- PR Procedure Summary

① Commit + Push Changes
② If necessary, rebase your feature branch on the develop branch
③ Create a PR
④ Discussion + Review, making extra commits if necessary
⑤ Reviewer approves, this creates a merge commit on the develop branch.