

Overview

This page contains the exercises for Part 1 (the first 5 weeks) of COMSM0085 Software Tools. Weeks usually contain two workbooks, and each workbook has associated videos to watch before the lab (on the main page for each workbook) as well as a set of practical exercises which you'll tackle before and during the lab.

Part 1 focuses on tools for setting up systems and writing code. We'll start with system administration tasks (like connecting to machines via ssh and updating software) and then move on to making good use of Unix tools, progressing to shell scripting and how to automate or simplify tasks you might encounter as a developer. Then we'll look at tools you can use to support programming in several different languages, including version control and debuggers. Finally, we'll cover how you can store structured information effectively in a relational database, and write queries to interact with that database.

System Administration

Videos

Before this activity, you should watch the following videos - most of them are quite short. The videos are hosted on Microsoft Stream, and you will need to authenticate via university log-in to watch them.

Video	Length	Slides
SSH	10 minutes	slides
Vagrant	12 minutes	slides
Package managers	6 minutes	slides

NB: The videos were recorded last year. Almost everything still applies, but you might see us talking about 'Alpine' or 'apk' -- the Linux VM we used to use, and its package manager. Look at the updated slides to see how Alpine-specific commands translate to Debian.

Exercises

- Secure shell
- Installing vagrant and Debian
- Debian system administration

Secure shell

Secure shell (SSH) is a protocol to allow you to remotely log in to another computer, such as a lab machine. Almost everyone who uses SSH uses the free OpenSSH implementation, which is standard on pretty much every Linux distribution and is also available for Windows and Mac - and even for the mobile operating systems iOS and Android.

We will see in more detail how SSH manages connections later on, but for now imagine that it opens a network connection between your own machine, and a shell running on a different machine. When you type something, SSH encrypts this and sends it to the other machine which decrypts it and passes it to the shell (or any other program you're running); when the shell replies then SSH encrypts that and sends it back to you. For this to work, (Open)SSH is actually two programs:

- `ssh` is the client, which you run on your machine to connect to another machine.
- `sshd` is the server, or *daemon* in UNIX-speak. It runs in the background on the machine you want to connect to, and needs to be installed by the system administrator.

Advanced note

SSH uses TCP port 22 by default.

Check your client

First of all, let's check that the ssh client is working.

- Open a terminal on your own machine: linux, mac OS and windows subsystem for linux should be fine. Windows 10 CMD might work too if you have the windows version of openssh installed (for example if you have git installed which uses ssh behind the scenes).
- Type `ssh localhost` and press ENTER. Several different things could happen:
 - If it asks for a password, then the ssh client is working, and a ssh server is running on your current machine. The password would be your user account password, but we don't actually want to log in again so cancel with Control+C.
 - If it succeeds without a password, then the client is working and a ssh server is running on your machine and either you do not have a password, or you already have a key set up. Type `exit` and press ENTER to get back to your previous shell.
 - If it shows "connection refused", then you have the ssh client correctly working but no server running on your own machine. This is not a problem, as we're trying to log in to the lab machines, so we need a client on our machine and a server on the lab machine.

- If it shows an error that ssh is not found, then you don't have (Open)SSH installed which is very unusual except on windows CMD - in which case please switch to using the windows subsystem for linux.

Connect to the lab

The lab machines have names `it#####.wks.bris.ac.uk` where the hashes represent a number from 075637 up to 075912. However, not all of them will be working at any one time, if everyone connects to the same machine then it will quickly get overloaded, and for security reasons the lab machines are not directly accessible from the internet. Instead, we will use two other machines to connect:

- The bastion host `seis.bris.ac.uk`. This is reachable over SSH from the internet, and is on a university network that lets you connect further to the lab machines. You should not attempt to do any work on seis itself, as most of the software you would like to use (like compilers) is not installed there. However, you do have a home directory on seis for storing things like SSH keys.
- The load balancer `rd-mvb-linuxlab.bristol.ac.uk`. This connects you to a lab machine that is currently running and ensures that if everyone uses this method to connect, then they will be more or less equally distributed among the running machines.

Try the following:

- On your terminal, type `ssh USERNAME@seis.bris.ac.uk` where you replace USERNAME with your university username, e.g. `aa20123`. Obviously, you will need a working internet connection for this.
- If it asks you whether you are sure, type `yes` and press ENTER. SSH will only do this the first time you connect to a machine that you have never used before.
- When prompted, enter your university password and press ENTER.
- You should now see the prompt on seis, which looks something like `-bash-4.2$`. Try the command `uname -a` to print information about the system (uname on its own prints the operating system name, `-a` shows "all" information). The reply line should start `Linux seis-shell`, which is the operating system and host name.
- On the seis prompt, type `ssh rd-mvb-linuxlab.bristol.ac.uk`. This might take a few seconds; say yes if it asks you if you're sure, then enter your password again when prompted. We didn't have to give a username again because you are already logged in to seis with your university username (`whoami` shows this) and when you ssh without giving a username, it uses the one you are currently logged in as.
- You should now be connected to a lab machine, with a prompt of the form `USERNAME@it#####:~$`.
- Try `whoami` and `uname -a` to check who you are logged in as, and where; also try `hostname` which just prints the machine name.

- Type `exit` twice to get back to your own machine. (Once gets you back to `seis`, twice closes the ssh connection completely.)

Connecting to one machine through another machine (in this case `seis`) as a proxy is such a common use case that ssh in fact has an option for it. Note however that this will not normally work from a windows CMD terminal, although it does work on Windows Subsystem for Linux (and on Mac and Linux).

```
ssh -J USERNAME@seis.bris.ac.uk USERNAME@rd-mvb-linuxlab.bristol.ac.uk
```

The `-J` for "jump through this host" even accepts a comma-separated list of hosts if you need to connect through more than one. However, you need to repeat your username for every machine.

You now know how to log in to a lab machine, but in both methods you had to type your password twice - let's make that easier. The answer is not to store your password in a file, but to use keys instead.

Setting up ssh keys

When you connect to a machine, the client on your computer and the daemon on the machine you're logging in to run a cryptographic protocol to exchange keys and set up a shared secret key for the session, so that what one side encrypts the other side can decrypt again. It also authenticates you to the server using one of several methods.

You might have heard from a security source that there are three main authentication factors: something you know (password or PIN), something you have (physical key, digital key, ID card, passport) and something you are (biometrics). An authentication method that requires two of these is called two-factor authentication and this is considered good security practice. For ssh, this means:

- You can log in with a username and password, that is "something you know". This is the default, but not the most secure.
- You can log in with a (digital) key, that is "something you have". This is more secure, as long as your key (which is just a file) doesn't get into the wrong hands, and also the most convenient as you can log into a lab machine or copy files without having to type your password.
- You can log in with a key file that is itself protected with a password. This gets you two-factor authentication.

The keys that SSH uses implement digital signatures. Each key comes as a pair of files:

- A private key (also known as secret key) in a file normally named `id_CIPHER` where CIPHER is the cipher in use. You need to keep this secure and only store it in places that only you have access to.

- A public key in a file normally named `id_CIPHER.pub`. You can share this with the world, and you will need to store a copy of it on any machine or with any service that you want to log in to (for the lab, because the lab machines all share a file system, you only need to store it once - but seis has a separate file system so you need a separate copy there).

Let's create a key pair:

- On your own machine, make sure you are not connected to a lab machine or seis, then type the command `ssh-keygen -t ed25519`. (If you get an "unknown key type" error, then you are using an outdated version of OpenSSH and for security reasons you should upgrade immediately.) *Note: type `ed25519` directly, do not replace this with your username. It stands for the "Edwards curve over the prime $2^{255}-19$ " cryptographic group, if you want to know.*
- When it asks you where to save the file, just press ENTER to accept the default, but make a note of the path - normally it's a folder `.ssh` in your home directory.
- If it asks you "Overwrite (y/n)", say no (n, then ENTER) as it means you already have a key for something else - either ssh directly or something that uses it, like github. Restart key generation but pick a different file name.
- When it asks you for a password, we recommend that you just press ENTER which doesn't set a password (good security, maximum convenience). If you do set a password, it will ask you to type it twice and then you will need the password and the key file to use this key (maximum security, less convenient).

Advanced note

The `-t` parameter selects the cryptographic algorithm to use, in this case `ed25519`, which is modern, peer-reviewed, and generally considered one of the most secure public-key algorithms available. However some older ssh versions don't accept `ed25519`.

If you ever need to use SSH keys to a machine that doesn't like `ed25519`, then use the key type "rsa" instead. We would recommend you avoid the alternatives "dsa" and "ecdsa" if at all possible as there is speculation among cryptographers that there may be a flaw in the design.

For example, although seis supports `ed25519`, the old cs bastion host `snowy.cs.bris.ac.uk` still uses an older version of SSH, so you would need to generate a `rsa` key to connect to that.

Have a look at the folder where your keys are stored. `ls -l ~/.ssh` will do this, unless you chose somewhere else to store them when you created them:

```
-rw-----. 1 vagrant vagrant  411 Oct  7 10:50 id_ed25519
-rw-r--r--. 1 vagrant vagrant   98 Oct  7 10:50 id_ed25519.pub
-rw-r--r--. 1 vagrant vagrant 1597 Oct  7 11:54 known_hosts
```

Note the permissions on these files in my example. The private key (first line) has a permissions line at the start of `(-) (rw-) (---) (---)` where I've added brackets to make clearer what is going on. The first bracket only applies to special file types (e.g. `d` for directory). Next, the owner permissions which are in this case read and write (the third one would be `x` if the file were an executable program). The last two brackets are the permissions for the group and for everyone else, and these are all off so no-one except yourself (and root) can read your key file. OpenSSH is picky about this and will refuse to use a private key that other people have access to.

The public key permissions are `(-) (rw-) (r--) (r--)` which means that the owner can read and write, and the group and everyone else (assuming they have access to the folder) can read the public key, which is fine. It's a public key after all.

`known_hosts` is where SSH stores the public keys of computers you've already connected to: every time you answer yes to an "Are you sure you want to connect?" question when you connect to a new computer for the first time, it stores the result in this file and won't ask you again the next time. The file format is one key per line and you can edit the file yourself if you want to.

Set up key access on SEIS

First, we need to upload our public key to the `~/.ssh` directory on seis. Even before this, we need to make sure the directory exists though:

- Log in to seis with ssh and your password.
- Try `ls -al ~/.ssh`. If it complains the folder doesn't exist, create it with `mkdir ~/.ssh`.
- Log out of seis again with `exit`.

The command for copying a file is `scp` for secure copy, which works like `cp` but allows you to include remote hosts and does the copy over SSH. Run this from your own machine:

```
scp ~/.ssh/id_ed25519.pub "USERNAME@seis.bris.ac.uk:~/.ssh/"
```

Obviously, replace `USERNAME` with your university username. This will ask for your password again. Note two things here: first, to set up access on seis, we are uploading the public key - not the private key! - and secondly, that we put double quotes around the destination. This is because the `~` character meaning home directory is handled by our shell, but we don't want our local shell to expand it, instead we want the shell on seis launched by `scp` to expand it to our home directory on that machine.

The general syntax of `scp` is `scp source destination` and source or destination may be of the form `[USERNAME@]HOSTNAME:PATH` - if it contains a colon (`:`), then it refers to a file on a different machine.

Now log in to seis over ssh and type your password one last time. Then run the following:

```
cd .ssh  
cat id_ed25519.pub >> authorized_keys  
chmod 600 authorized_keys
```

SSH will accept a public key if it is listed in the file `authorized_keys` in the user's `.ssh` folder, the format is one line per key. Instead of just copying `id_ed25519.pub` to `authorized_keys`, which would overwrite the latter file if it already existed, we use the construction `cat SOURCE >> DEST` to have our shell append the source file to the destination. *Note: it's **authorized**, the American spelling -- if you use the British spelling here OpenSSH won't know to read that file.*

However, if the authorised keys file didn't exist already, then it has now been created with default permissions and SSH won't accept that for security reasons. `chmod` means change permissions (also known as "mod bits") and 600 is the bit pattern we want in base 8, because that is how permissions work for historical reasons. Permissions are a bitfield of 9 bits, the first three are read/write/execute for the owner, the next three the same for the group, and then for everyone else. If you `ls -l` you will see this in a slightly more human-readable format, namely `rw-----` where a minus means that a bit is turned off.

Now type `exit` to get back to your own machine, and then `ssh USERNAME@seis.bris.ac.uk` to log back in to seis. It should log you in without asking for a password, and you have now set up key-based SSH authentication for seis.

Note: if you set a password on your SSH key earlier, then it will ask you for a password, and it will expect the key password not your uni password. You know not to ever reuse your uni password for anything else, right?

If for some reason something doesn't work with ssh, the first thing to try is to add the `-v` switch enable debugging information (you can even do `-vv` or `-vvv` to see even more detail, but we don't need that). If there is a problem with the permissions on your private key file for example, then you will see SSH complain in the debugging information.

Setting up keys for lab machines

You can now get into seis with a key, but you want to be on a lab machine to get work done.

To connect from your machine to seis, you need a private key on your machine and a public key on seis. To connect from seis to a lab machine, it would seem like you need a public key on the lab machine and a private key on seis. You do not want to upload your private key to seis though for security reasons, so instead we are going to use a SSH feature called *agent forwarding* which means that if you SSH into one machine, then when you try and SSH

further into another machine SSH will reuse the same key. The way to do this is to use the `-A` command line flag.

Advanced note

The point of key-based authentication is that your private key never leaves your own machine, so even university administrators never get to see it, which would not be guaranteed if you stored a copy on a university machine.

Logging in to a machine does not send the key to that machine. Instead, the machine sends you a challenge - a long random number - and SSH digitally signs that with the private key on your own machine, and sends the signature back which the remote machine can verify with the public key. Seeing a signature like this does not let the machine create further signatures on your behalf, and it definitely does not reveal the key.

What agent forwarding does is it allows challenges and signatures to be forwarded across multiple connections, but the key never leaves your own machine.

This way, you can create one SSH key and use it for university, github and anything else that you access over SSH, and even if one service is breached then this does not give the attacker access to your accounts on the other services.

- Log in to seis with `ssh USERNAME@seis.bris.ac.uk`. You should not need a password anymore.
- Log in to the lab machines with `ssh rd-mvb-linuxlab.bristol.ac.uk` and enter your password. Check that the `~/.ssh` folder exists and create it if it doesn't, as you did before on seis, then `exit` again to seis.
- Copy your public key file from seis to the lab machines with `scp ~/.ssh/id_ed25519.pub "rd-mvb-linuxlab.bristol.ac.uk:~/.ssh/"`. This will ask for your password again.
- Log in to a lab machine with `ssh rd-mvb-linuxlab.bristol.ac.uk` and enter your password one last time. On the lab machine, install the public key with the following:

```
cd .ssh  
cat id_ed25519.pub >> authorized_keys  
chmod 600 authorized_keys
```

- Log out of the lab machine and seis again by typing `exit` twice.

The steps above were necessary because your home directory on seis is not the same as on the lab machines. However, your home directory is the same across all lab machines, so you don't need to install the key on each one separately. You might have noticed that when copying or `ssh`-ing from seis to the lab machines, you don't have to repeat your username: this is because it is the same on all these machines.

From now on, from your own machine, you should be able to get directly into a lab machine with the following command, which should not ask for your password at all:

```
ssh -A -J USERNAME@seis.bris.ac.uk USERNAME@rd-mvb-linuxlab.bristol.ac.uk
```

Unfortunately, `-J` will not work on a windows CMD terminal, although it should work on Windows Subsystem for Linux. Once we have set up a configuration file, there will be a way to work around this problem. Mac and Linux users should be fine though, as should anyone running these commands from a Linux VM on their own machine, whatever their host OS.

Setting up a configuration file

You now have a log in command that works, but you still have to type a lot, and you need to type your username twice. We can improve this by using a configuration file.

SSH reads two configuration files: one for all users at `/etc/ssh/ssh_config` (`/etc` is where POSIX programs typically store global settings) and a per-user one at `~/.ssh/config`. The site <https://www.ssh.com/ssh/config/> or just `man ssh_config | less` on a terminal contain the documentation (`man` means manual page, and `less` is a program that shows a file on page at a time and lets you scroll and search).

Create a file called simply `config` in your `.ssh` directory on your own machine. You can do this for example with `touch config` (make sure you're in the `.ssh` directory first, `cd ~/.ssh` gets you there), and then editing it in your favourite text editor. Add the following lines, replacing `USERNAME` with your username twice:

```
Host seis
  HostName seis.bris.ac.uk
  User USERNAME
```

```
Host lab
  HostName rd-mvb-linuxlab.bristol.ac.uk
  ProxyJump seis
  User USERNAME
```

This now lets you use simply `ssh lab` to log in to a lab machine via `seis` (agent forwarding is implied when you use `ProxyJump`), or `ssh seis` if you want to access `seis` directly for example to update your keys there.

- Try `ssh lab` from your own machine. This will be your main way to log in to a lab machine from now onwards.

Advanced note

If you want to learn another useful skill as you go along, here is one way to edit files on the command line. Many linux distributions have an editor called `nano` built in which runs in the terminal, so `nano config` edits the file called `config` (creating it if it doesn't exist, when you save for the first time). It is fairly self-explanatory, the command `Control+X` quits

as you can see on the command bar at the bottom of the screen in nano and if you quit with unsaved changes, it will ask you if you want to save.

Nano is installed on SEIS and on the lab machines, and within the Debian distro you will install in Vagrant, so you can use it to edit a file remotely.

And something for Windows users:

If you are on Windows and are using OpenSSH through a CMD terminal, a bug in OpenSSH prevents the `-J` option from working. However, you can write your file like this instead:

```
# ~/.ssh/config file for WINDOWS CMD users only

Host seis
  HostName seis.bris.ac.uk
  User USERNAME

Host lab
  HostName rd-mvb-linuxlab.bristol.ac.uk
  ProxyCommand ssh.exe -W %h:%p seis
  User USERNAME
```

This should get `ssh lab` to work for you as well.

Using different keys

You do not need this for the lab, but if you are ever managing different systems and accounts then you might use a different key file for each one. In this case, ssh on the command line lets you do `-i FILENAME` to select a private key file, and in the configuration file you can select a file for a particular host with the `IdentityFile FILENAME` line. By default, ssh will search for files in `.ssh` with names `id_CIPHER`, as you can see if you launch a connection with the `-vv` parameter which shows detailed debugging information.

Installing Vagrant and Debian

Vagrant is a program to manage virtual machines (VMs). Based on a configuration file called a `Vagrantfile`, it can download and configure disk images, which it calls boxes, and call other programs to run them. Vagrant does not run the VM by itself, so you will need another program like `virtualbox` for that.

Installing on your own machine

To use vagrant on your own machine (recommended), follow these steps:

- Go to <https://www.vagrantup.com/downloads> and download the version of vagrant for your operating system. Windows, Mac OS and common versions of Linux are supported.
- Download and install Virtualbox from <https://www.virtualbox.org/wiki/Downloads>.
- Reboot your machine.

If you are on Linux, you can of course also install the programs from your distribution's repository. Vagrant's developers actually recommend against this because they claim that some distributions package outdated versions, but it is your choice.

Configuring a box

Next, you are going to configure a virtual machine using `Debian linux`, a Linux distribution that we will be using in this unit.

- Create an empty folder somewhere.
- In that folder, create a file called `Vagrantfile` (capitalised, and with no extension) and add the following lines to it - or just download the file from [here](#):

```
Vagrant.configure("2") do |config|
  config.vm.box = "generic/debian12"
  config.vm.synced_folder ".", "/vagrant"

  config.vm.provision "shell", inline: <<-SHELL
    echo "Post-provision installs go here"
  SHELL
end
```

This configuration file is actually a script in the ruby programming language, but you don't need to learn that to use vagrant. Let's look at what it does.

- `config.vm.box` selects the virtual machine image, or box in vagrant-speak, to use. You can see a list of available ones at <https://app.vagrantup.com/boxes/search>.
- `config.vm.synced_folder` sets up a shared folder between the guest (virtual machine) and host (your machine).
- The `config.vm.provision` runs a provisioning command when the box is first downloaded and installed. These commands run as root on the virtual machine, and in this case we are using the `apt` package manager (we will talk about this later on) to install the packages `git`.
- The `<<-SHELL` construction is called a "here document", and is a way in some programming languages of writing multi-line strings. It tells ruby to treat everything until the closing keyword `SHELL` (which is arbitrary) as a string, which can contain several lines.

Running vagrant

- Open a terminal in the folder containing the Vagrantfile. If you are on windows, both the windows CMD and the windows subsystem for linux terminal will work equally well for this purpose.
- Run the command `vagrant up`. This starts the virtual machine configured in the current folder, and if it has not been downloaded and provisioned yet (as is the case when you run `up` for the first time) then it does this for you as well.
- When vagrant tells you the machine is running, run `vagrant ssh` to log in to your virtual machine. If it asks you for a password, use `vagrant`.
- You should now see the virtual machine prompt `vagrant@debian12:~$`. Try the command `ls /` and check that there is a folder called `vagrant` in the top-level folder, along with system ones with names like `usr` and `bin`.

There are two kinds of errors you might get during `vagrant up`:

- If vagrant complains that it can't find a provider, then you have probably not installed `virtualbox`, or not rebooted since installing it.
- If you get some odd crash or error message about hypervisors, see the page <https://www.vagrantup.com/docs/installation> for instructions, section *Running Multiple Hypervisors*. Basically, you cannot run vagrant when another program is already using your processor's virtualisation subsystem, and the page gives instructions how to turn off the other one.

Shutting down cleanly

To exit the virtual machine, type `exit` which will get you back to the shell on the host machine. On the host, `vagrant halt` cleanly shuts down the virtual machine.

Promise yourself that you will always do this before turning off your computer, if you have been using Vagrant!

Running on a lab machine

Vagrant is already installed on the lab machines in MVB 2.11, so you can remotely log in and launch a box from there. This will get you exactly the same Debian environment as when you run it on your own machine, and everyone should try this out too. If for some reason you cannot run Vagrant on your machine, then as long as you have an internet connection you should still be able to run it on the lab machines.

First, we connect to a lab machine: open a terminal and run the command `ssh lab` that you configured in the previous exercise on SSH.

On the lab machine, we need to create a folder and load a Vagrantfile as above, but let's download the Vagrantfile from the unit webpage instead of typing it out. Run the following shell commands (the third one starting `wget` must be all on one line, even if your web browser has added a line break):

```
mkdir softwaretools  
cd softwaretools  
wget https://raw.githubusercontent.com/cs-uob/COMSM0085/master/exercises/part1/src/resources/Vagrantfile
```

You can call the top folder (softwaretools) anything you like and put it anywhere you want. You can now run `vagrant up` followed by `vagrant ssh` from inside that folder.

Advanced note

When you `vagrant up`, vagrant internally connects port 22 on the guest (which `sshd` on the guest is listening to) to port 2222 on the host. When you provision a vagrant machine, this creates a key pair on the host and loads the public key into the guest. The private key is actually in the file `.vagrant/machines/default/virtualbox/private_key` on the host, and the public key in `/home/vagrant/.ssh/authorized_keys` on the guest. So what `vagrant ssh` does is launch `ssh -i KEYFILE vagrant@localhost -p 2222`.

Warning about lab machines - read carefully!

Your files in your home directory on a lab machine are stored in a network folder, so that you see the same files whichever lab machine you log in to; they are also automatically backed up.

If lots of students created lots of VMs in their home folders, this would take up lots of space, and it would be slow: running an operating system over a network share causes both bandwidth and latency problems.

Instead, IT has configured vagrant on the lab machines to store VMs in the `/tmp` folder which is local to each machine. This means that:

- If you log in to a different lab machine, your VMs will be gone.
- If you log in to the same lab machine but it has restarted since you last logged in, your VMs will be gone.
- Your VMs, and with them any files you store in the VM itself, are not backed up.

This is not as much a problem as it seems because this is how virtual machines are meant to work: if one is not available, vagrant downloads and provisions it. For this reason, for any software you want installed on your VMs in the lab machines, you should write the install command into the provisioning script in the Vagrantfile so it will be re-installed the next time Vagrant has to set up the VM. We will learn how to do this soon.

However, this still leaves files that you create on the VM itself, such as the ones you will create for the exercises in this unit. The basic warning is that *any files in your home directory will be lost when the VM is rebuilt*. That is why we have set up a shared folder which you can access as `/vagrant` on the VM, which maps to the folder containing your Vagrantfile on the host machine. Because this is stored under your home folder on the lab machine, it lives on the network file store and so it is backed up and available from all lab machines.

So whenever you log in to a VM on a lab machine to do some work, you should `cd /vagrant` and use that instead of your home folder for any files that you don't want to lose. If you are running vagrant on your own computer, then nothing in the VM will be deleted unless you give Vagrant a command to destroy or rebuild the VM yourself.

Debian system administration

Start your Debian box if necessary by going to the folder with the `Vagrantfile` in your terminal, and typing `vagrant up`. Log in to your Debian box with `vagrant ssh`. We are going to get to know Linux in general and Debian in particular a bit.

The file system

Linux (and other POSIX-like operating systems) work with a single file hierarchy with a root folder `/`, although there may be different file systems mounted at different places under that. How files are organised in here are documented in the [Filesystem Hierarchy Standard \(FHS\)](#). Have a look with the command `ls /`:

`/bin` stands for binaries, that is programs that you can run. Have a look with `ls /bin`: there will be a lot of commands in here, including `ls` itself. Indeed you can find out where a program is with `which`, so `which ls` will show you `/usr/bin/ls` for example.

`/usr` is a historical accident and a bit of a mess. A short history is on [this stackexchange question](#) but essentially, in the earliest days,

- `/bin` was only for binaries needed to start the system - or at least the most important binaries that needed to live on the faster of several disk drives, like your shell.
- `/usr/bin` was where most binaries lived which were available globally, for example across all machines in an organisation.
- `/usr/local/bin` was for binaries installed by a local administrator, for example for a department within an organisation.

In any case, `/usr` and its subfolders are for normally read-only data, such as programs and configuration files but not temporary data or log files. It contains subfolders like `/usr/bin` or `/usr/lib` that duplicate folders in the root directory. Debian's way of cleaning this mess up is to make its `/bin` just a link to `/usr/bin` and putting everything in there, but in some distributions there are real differences between the folders.

If you have colours turned on (which is the default) you will see some files are green, but others are blue - this indicates the file type, green is an executable program, blue is a link to another file. Have a look with `ls -l /bin`: the very first character of each line indicates the file type, the main ones being `-` for normal file, `d` for directory and `l` for a so-called *soft link*. You can see where each link links to at the end of this listing. For example, `slogin` links to `ssh`. Other links point at files stored elsewhere in the filesystem -- you'll see a lot of references to `/etc/alternatives/`.

`/etc` stores system-wide configuration files and typically only root (the administrator account) can change things in here. For example, system-wide SSH configuration lives in `/etc/ssh`.

`/lib` contains dynamic libraries - windows calls these `.dll` files, POSIX uses `.so`. For example, `/lib/x86_64-linux-gnu/libc.so.6` is the C library, which allows C programs to use functions like `printf`.

`/home` is the folder containing users' home directories, for example the default user vagrant gets `/home/vagrant`. The exception is root, the administrator account, who gets `/root`.

`/sbin` (system binaries) is another collection of programs, typically ones that only system administrators will use. For example, `fdisk` creates or deletes partitions on a disk and lots of programs with `fs` in their name deal with managing file systems. `/sbin/halt`, run as root (or another user that you have allowed to do this), shuts down the system; there is also `/sbin/reboot`.

`/tmp` is a temporary filesystem that may be stored in RAM instead of on disk (but swapped out if necessary), and that does not have to survive rebooting the machine.

`/var` holds files that vary over time, such as logs or caches.

`/dev`, `/sys` and `/proc` are virtual file systems. One of the UNIX design principles is that almost every interaction with the operating system should look to a program like reading and writing a file, or in short *everything is a file*. For example, `/dev` offers an interface to devices such as hard disks (`/dev/sda` is the first SCSI disk in the system, and `/dev/sda1` the first partition on that), memory (`/dev/mem`), and a number of pseudoterminals or ttys that we will talk about later. `/proc` provides access to running processes; `/sys` provides access to system functions. For example, on some laptop systems, writing to `/sys/class/backlight/acpi_video0/brightness` changes the screen brightness.

The `/vagrant` folder is not part of the FHS, but is our convention for a shared folder with the host on Vagrant virtual machines.

Package managers

Linux has had package managers and repositories since the days when it was distributed on floppy disks. A repository is a collection of software that you can install, and can be hosted anywhere - floppy disk, CD-ROM, DVD or nowadays on the internet. A package manager is software that installs packages from a repository - so far, this sounds just like an *app store* but a package manager can do more. For one thing, you can ask to install different versions of a particular package if you need to. But the main point of a package manager is that packages can have dependencies on other packages, and when you install one then it installs the dependencies automatically.

To illustrate this, we're going to go on a tour of one kind of software with which you'll want to get very familiar: text editors that work in the console. Text editing is fundamental to a lot of system administration tasks as well as programming, and people often find that a familiar editor becomes a favourite tool.

Two console-based text editors are already installed in Debian: `nano` and `vim`.

`nano` is a basic text editor that works in the console, and is installed in most Linux distributions including the ones on seis and the lab machines, so you can use it to edit files remotely. You can type `nano FILENAME` to edit a file. The keyboard shortcuts are at the bottom of the screen, the main one you need is Control+X to exit (it will ask if you want to save, if you have unsaved changes). Nano is considered fairly friendly as console editors go.

`vim` is the 1991 improved version of the even older (1976) `vi` editor (which is also installed). It is a modal editor with a steep learning curve, but its extremely powerful editing grammar and widespread availability mean it regularly appears towards the top of lists of favourite text editors. If you want to get started with vim, I suggest you start by typing `vimtutor` at the commandline -- this opens vim with a file that guides you through basic vim usage.

Another console-based editor dating from the mid-70s is `emacs`, `vi`'s traditional Lisp-based rival. However, `emacs` is not installed by default: type `emacs` at the console and you will get `emacs: command not found`. You can install it with the command

```
sudo apt install emacs-nox
```

- `sudo` (superuser do) allows you to run a command as root, also known as the administrator or superuser. Depending on how your system is configured, this might be not allowed at all (you can't do it on the lab machines), or require a password, but on the Vagrant box you're using you are allowed to do this. It is good practice to use `sudo` for system administration instead of logging in as root directly, but if you ever really need a root shell then `sudo bash` gets you one - with `#` instead of `$` as prompt to warn you that you are working as root.
- `apt` is the Debian package manager.
- `install PACKAGE` adds a package, which means download and install it and all its dependencies (you'll see a list of these to confirm you want to install them -- you do).

We're installing `emacs-nox` because this is the version of `emacs` packaged for use from the console. If you tried to install just `emacs` then the package manager would identify that you need a graphical display manager to run `emacs`'s GUI mode and install a lot more dependencies to enable that, which we don't need.

Now that `emacs` is installed, you can launch it with `emacs`, and from there you should see some introductory instructions, including how to access an `emacs` tutorial which will teach you how to use it. (If you want to exit `emacs`, Control-X followed by Control-C should do it).

Other popular editors include

- `mcedit`, a file editor which comes as part of the 'midnight commander' package, which you can install with `sudo apt install mc`. Launch the editor with `mcedit filename` and test it out (Alt-0 to exit).
- `tilde`, an editor with a GUI-esque menu system controlled through Alt-letter sequences. Install with `sudo apt install tilde`.
- `micro`, a simple editor somewhat like an advanced version of `nano`. Install with `sudo apt install micro`.

You can also find out information about packages with `apt info PACKAGE` -- try this for one of the above.

I suggest that you try out some of these editors, and figure out how you prefer to edit files. Some of these tools might require time investment to learn, but doing this early in your CS career could be a good decision.

Whichever editor you end up deciding to use, you probably won't need to keep all the alternatives installed. You can leave `nano` and `vim` installed, but for the other editors you've tried out above and decided you don't like, you can (and should) remove them from the system with `sudo apt remove PACKAGE`.

Update and upgrade

The repositories that you are using are recorded in `/etc/apt/sources.list`, have a look at this file with `cat` or `nano` to see where they are, then look up the sites in your browser. There are folders for different Debian versions and package index files for different architectures.

Two commands a system administrator should run regularly for security reasons:

- `sudo apt update` fetches the new package list from the repository. This way, apt can tell you if any packages have been updated to new versions since you last checked.
- `sudo apt upgrade` upgrades every package that you already have installed to the latest version in your local package list (downloaded when you do an `apt update`).

Lab machines

If you are running a virtual machine on the lab machines, then your virtual machine might not be around after the lab machine reboots or you log out and in again and end up on a different machine - as the notice when you log in tells you, the virtual machines are stored under `/tmp`.

It would be annoying to have to reinstall your favourite packages (like your chosen text editor) every time you log in to a different machine, so you should put them in your Vagrantfile and then `vagrant up` will do this for you automatically. The Vagrantfile already contains a line `echo`, you can put an `apt install PACKAGE` line below this, and can list as many packages as you like. There is no `sudo` here because when Vagrant is installing the system, it is running as root automatically.

- Unless it is `nano` or `vim`, add the package for your favourite text editor to this line so next time you rebuild the Vagrant machine, they are added automatically.
- Log out of your vagrant machine and do a `vagrant destroy` which removes the virtual machine. Then reload with `vagrant up` which will download and provision the box again.
- Log in with `vagrant ssh` and check that the editor is installed.

The POSIX Shell

Videos

Video	Length	Slides
The shell	30 minutes	slides
Pipes 1	20 minutes	slides
Pipes 2	30 minutes	slides

Exercises

- Shell expansion
- Pipes
- Regular expressions

Shell expansion

This exercise is about studying shell expansion. You should run it on your Debian VM in Vagrant.

Create a C program `arguments.c` with the following contents. You can use `nano arguments.c` for this, for example.

```
#include <stdio.h>

int main(int argc, char** argv) {
    for(int i=0; i < argc; i++) {
        printf("Argument #%i: [%s]\n", i, argv[i]);
    }
    return 0;
}
```

Compile this with `gcc -Wall arguments.c -o arguments`.

Whitespace

The program prints all its arguments, one per line. The program gets its arguments from the program that started it - in this case the shell. Try running the program with the following commands:

```
./arguments
./arguments hello
./arguments one two three
```

Now that you are familiar with what the program does, try the following:

```
./arguments one      two
./arguments "one two"
./arguments "one      two"
```

How, based on these examples, does the shell handle whitespace in the line you type?

Pattern matching

Try the following:

- `./arguments *` in the folder that contains the arguments program, and its source code `arguments.c`.
- Make an empty subfolder with `mkdir empty`, switch to it with `cd empty` and then run `../arguments *`. Since you are now in the subfolder, we need two dots at the start to say "run the program arguments in the folder *above*". What happens?
- Go back to the folder with the program by running `cd ..` and then do `ls` to check you're back in the right folder. In this folder, find three different ways to get the program to produce the following output:

```
Argument #0: [./arguments]
Argument #1: [*]
```

Files with spaces in their names

The command `touch FILENAME` creates a file. Create a file with a space in its name by typing `touch "silly named file"`. What would happen if you left the quotes off (you can try it, then do `ls`)?

Start typing `ls sill` and then press TAB to autocomplete. Assuming you have no other files whose name starts with *sill*, what happens? Use this method to get the arguments program to print the following:

```
Argument #0: [./arguments]
Argument #1: [Hello world!]
```

The command `rm` (remove) deletes files again. Use it to remove your file with spaces in its name, using one of several methods to get the shell to pass the spaces through to `rm`.

Shell variables

In the shell, `VARIABLE=VALUE` sets a variable to a value and `$VARIABLE` retrieves its value. For example, to save typing a filename twice:

```
p=arguments
gcc -Wall $p.c -o $p
```

which expands to `gcc -Wall arguments.c -o arguments`. If you want to use a variable inside a word, you can use curly braces: `${a}b` means the value of the variable `a` followed by the letter `b`, whereas `$ab` would mean the value of the variable `ab`.

It is good practice to double-quote variables used like this, because if you tried for example to compile a program called `silly name.c` with a space in its name, then

```
program="silly name"  
gcc -Wall $program.c -o $program
```

would expand to

```
gcc -Wall silly name.c -o silly name
```

and this would confuse your compiler because you are telling it to compile three source files called `silly`, `name.c` and `name` to a program called `silly`. Correct would be:

```
program="silly name"  
gcc -Wall "$program.c" -o "$program"
```

which expands to

```
gcc -Wall "silly name.c" -o "silly name"
```

which does what you want - if you indeed want a program with a space in its name!

There is no harm in double-quoting a shell variable every time you want to use it, and this is good practice as it still works if the variable is set to a value that contains spaces.

Note that we also had to quote setting the variable name in the first place, because

```
program=silly name
```

would translate as: set the variable `program` to the value `silly`, then execute the program `name`. Variable assignments only apply to the first argument following them, although you can assign more than one variable.

Note that this does not work as expected either:

```
file=arguments gcc -Wall "$file.c" -o "$file"
```

The problem here is that the shell first reads the line and substitutes in the value of `$file` (unset variables expand to the empty string by default) before starting to execute the command, so you are reading the variable's value before writing it. Leaving off the quotes doesn't help: you need to set the variable on a separate line.

Pipes

The command `ls | head` runs ls and head and pipes the standard output of ls into the standard input of head.

The following shell commands are particularly useful in pipes:

- `cat [FILENAME [FILENAME...]]` writes the contents of one or more files to standard output. This is a good way of starting a pipe. If you leave off all the filenames, cat just reads its standard input and writes it to standard output.
- `head [-n N]` reads its standard input and writes only the first N lines (default is 10 if you leave the option off) to standard output. You can also put a minus before the argument e.g. `head -n -2` to *skip the last 2 lines* and write all the rest.
- `tail [-n N]` is like head except that it writes the last N lines (with a minus, it skips the first N ones).
- `sort` reads all its standard input into a memory buffer, then sorts the lines and writes them all to standard output.
- `uniq` reads standard input and writes to standard output, but skips repeated lines that immediately follow each other, for example if there are three lines A, A, B then it would only write A, B but if it gets A, B, A it would write all three. A common way to remove duplicate lines is `... | sort | uniq |`
- `grep [-iv] EXPRESSION` reads standard input and prints only lines that match the regular expression to standard output. With `-i` it is case-insensitive, and with `-v` it only prints lines that do *not* match the expression.
- `sed -e COMMAND` reads lines from standard input, transforms them according to the command and writes the results to standard output. `sed` has its own command language but the most common one is `s/SOURCE/DEST/` which changes substrings matching the source regular expression into the destination one.
- `wc [-l]` stands for word count, but with `-l` it counts lines instead. Putting a `wc -l` on the very end of a pipe is useful if you just want to know how many results a particular command or pipe produces, assuming the results come one per line.

All these commands actually take an optional extra filename as argument, in which case they read from this file as input. For example, to display the first 10 lines of a file called `Readme.txt`, you could do either `cat Readme.txt | head` or `head Readme.txt`.

Word list exercises - pipes and regular expressions

Most Linux distributions (including Debian) come with a dictionary file `/usr/share/dict/words` that contains a list of English words in alphabetical order, for use in spell-checking programs. The list includes a selection of proper nouns, for example

countries and cities. If you want to look at it on a system that doesn't have it, you can download it with:

```
wget https://users.cs.duke.edu/~ola/ap/linuxwords -O words
```

`wget` is one of two utilities for downloading files, the other being `curl`. Note that the option for output file name is a capital O, not a lowercase o or a zero.

Find one-line commands, possibly with pipes, to print the following to your terminal. You can either start each command with `cat /usr/share/dict/words | ...` or do it without `cat` by providing the `words` file as an argument to the first command in your pipeline.

If English is not your native language, ignore the guessing part - it is not assessed.

- The first word in the file. *Can you guess what it will be?*
- The last word in the file. *Can you guess this one?*
- The number of words in the `words` file - there is one word per line.
- The 6171st word in the file. *Can you read my mind and guess this word directly?*
- All words containing the letter Q, capitalised. (A regular expression containing a string of one or more letters matches all strings that contain the expression as a substring.)
- All words starting with the letter X. The regular expression `x` would match an X anywhere in the word, but `^x` matches an X only at the start of the string.
- All words ending in j. (The expression `'j$'` matches a j only at the end of the string, but you have to single-quote it to stop the shell from interpreting the dollar sign).
- The number of words containing the letter Q, ignoring case (e.g. capitalised or not).
- The first five words containing the letter sequence `cl`.
- All words containing the sequence "kp", but not "ckp". *Can you guess any of these?*
- The last 15 words of exactly two letters. The expression `.` (period) matches a single character, and `'^...$'` for example would match all strings of the format *exactly three characters between start and end of string*. You need to quote it because of the dollar sign.
- All words from the first 100 words on the list, which contain the letter y.
- The first five words that are among the last 100 words on the list, and contain the letter y (whether capitalised or not).
- All three-letter words with no vowels (aeiou). The regular expression `'[aeiou]'` matches any string that contains one of the bracketed characters; you need quotes to stop the shell from interpreting the brackets. Remember to exclude words with capitalised vowels as well. *There are 343 of these.*
- All words of exactly 7 letters, where the third one is an e and the word ends "-ded". *This kind of search is really useful for crosswords. There are 14 words of this form, can you guess them?*

Bonus regular expression question:

- Find all words that start with a P (whether capitalised or not), and contain at least four instances of the letter a. Putting a `*` after something in a regular expression searches

for any number of repetitions of this, including 0 so for example 'a*' would find words with any number of the letter a, including 0 (which is not what you want here). You need single quotes to stop the shell from expanding the *. Can you guess the words? There are 14 hits in the solution but essentially five words: two demonyms and three nouns which are not proper nouns, all with possessive and plural forms (bar one which is its own plural).

Regular expressions

For this exercise you'll want to refer often to a manual for `grep`. You can access one on the commandline by invoking `man grep`. You've already tackled some problems involving regular expressions in a previous exercise. Here are some more advanced questions that will require you to understand more about `grep`, its options, and how regular expression syntax works.

1. Study the documentation for the `-w` option. Contrive a file such that `grep PATTERN FILE` returns two different lines but `grep -w PATTERN FILE` returns only one line.
2. You'll have seen beforehand that you can count the results of a search with `grep PATTERN FILE | wc -l`. However, `grep` also has a `-c` option which counts matches. Can you find the situation where the `wc -l` approach and the `-c` approach produce different results? Can you explain why?
3. Some words have different spelling between British English and American English. For example, 'encyclopaedia' is valid in British English but not American. Can you write a regular expression that would match both of these words, but nothing else? How about matching both 'color' (American) and 'colour' (British)?
4. UK postcodes follow a general schema of two letters followed by one number, followed by an optional space, then another number, followed by two more letters. Can you write a regular expression that would match such sequences?
5. In practice, the above is a simplified version of the system, and a better UK postcode validator regex is known to be `^(([A-Z]{1,2}[0-9][A-Z0-9]?)|ASCN|STHL|TDCU|BBND|[BFS]IQQ|PCRN|TKCA) ?[0-9][A-Z]{2}|BFPO ?[0-9]{1,4}|(KY[0-9]|MSR|VG|AI)[-]?[0-9]{4}|[A-Z]{2} ?[0-9]{2}|GE ?CX|GIR ?0A{2}|SAN ?TA1)$`. Try breaking apart this monster to understand what is being tested, and find an example that would match the schema described for the fourth question but fail to match this expression.

Activity: Git

Videos

Video	Length	Slides
Git, part 1	34 minutes	slides
Git, part 2	27 minutes	slides
Git, part 3	19 minutes	slides

Exercises

- Git

Reading

Optional (but recommended).

- `man 7 giteveryday`

Git

Git is the defacto standard version control system used throughout CS. It also lives up to its name as being a *pain* to use. Take it slow and take it as a chance to practice using the Git commandline.

This lab *should* work fine on pretty much any computer (well... maybe not Windows) but we'd still recommend completing it in a virtual machine (mostly because we can test the lab in a virtual machine...).

If you'd like a `Vagrantfile` to base your work on you can use the following:

```
Vagrant.configure("2") do |config|
  config.vm.box = "generic/debian12"
  config.vm.synced_folder ".", "/vagrant"

  config.vm.provision "shell", inline: <<-SHELL
    apt-get update -y
    apt-get install -y git git-man apt-file
    apt-file update
  SHELL
end
```

Git documentation

Git comes with *extensive* documentation. Run:

```
apropos git
```

To see all of it, or run:

```
apropos git -a tutorial
```

To find documentation relating to git and (the `-a`) tutorials. Read any of the documentation you think might be useful with the `man` command.

Task: There is a man page that documents the *everyday git* commands you might want to use. Find it with `apropos` and read it with the `man` command.

You might also want to read the `gittutorial` pages...

Configuring your identity

Git is all about tracking changes to source code. That means it needs to know *who* made *what* changes.

Run the following two lines to set up git correctly. You only need to do this once when you install git, but not every time you create a new repository.

```
git config --global user.name "YOURNAME"  
git config --global user.email "YOUREMAIL"
```

The name and email address aren't actually sent anywhere or checked... they're just listed with alongside the changes you make to the code so if somethings wrong later programmers know who to blame (see `man git-blame`). You can put anything you like here (git will happily accept – as your email address, and it does not send you email).

This alters the *global* git configuration (the settings applied to *every* git repository you work on), but you can also make these changes on a repository by repository basis. Just drop the `--global` and run the command inside the git repository you want to apply the changes to. This is useful if you're *Bruce Wayne* and need to keep your public and private development projects separate (or if you do subcontracted development work).

For those of you using Vagrant

If you are running a VM on a lab machine, then you would need to reconfigure git every time vagrant rebuilds the VM, for example when you log in to a different lab machine. You can put these commands in your `Vagrantfile`, like anything else that you want to run when vagrant (re)builds your box, but they need to be run as the vagrant user and not the root user. So add the following block to your `Vagrantfile` just before the `end` line, editing your name and email address obviously. Normally vagrant will run these provision blocks as the system administrator `root`, but you can run it as the normal `vagrant` user by adding the `privileged: false` keyword.

```
config.vm.provision :shell, privileged: false, inline: <<-SHELL  
  git config --global user.name "YOURNAME"  
  git config --global user.email "YOUREMAIL"  
SHELL
```

If you start getting errors about the git command not being installed: install it! If you're using the Debian-based VM the command you need is `apt` (see `man apt` if you're not familiar with it).

Some people find having two provisioning blocks a bit messy. You could reduce them to just one block, but you'll need to use the `su` command to ensure that you configure git as the `vagrant` user.

A sample project and repository

Let's say you want to start developing a C program. Let's make a folder:

```
mkdir project1  
cd project1  
git init
```

The last command created an empty git repository in a subfolder called `.git`. We can check with `git status` to see whether there are any changes, and git reports `nothing to commit`.

Create a file `main.c`, with your text editor and add some sample content like this (you should be able to copy-paste into your terminal):

```
// file: main.c  
#include <stdio.h>  
  
int main() {  
    puts("Hi");  
    return 0;  
}
```

Advanced note

Which text editor should you use? It doesn't honestly matter, but it needs to be able to write plain text documents (those without formatting---so not Microsoft Word).

The *standard* editor is called `ed`: but don't use it! It is designed for back before computers had screens.

The *traditional* programmer editors are `vim` or `emacs`. You *should* learn one of them (Jo and Matt would both recommend `vim`) but they're rather confusing if you've never used a programmers' editor before. Absolutely worth it though; just expect to be rather slow at typing things for a week or two.

Easier editors include `nano` which works on the command line, `gedit` which is a bit more graphical or Microsoft's `Visual Studio` which can edit code on remote systems. They're all configurable and you can make all of them do things like syntax highlighting and code completion.

We don't care what editor you use: but make sure you make it work for you by configuring it. You're going to be spending an awful lot of your degree writing code: a bit of investment in text editor now will pay dividends later on!

(and I think everyone judges people who are *still* using `nano` in their final year...)

Do a `git status` and you will see `main.c` in red under *untracked files* - this is a new file that git does not know about yet. Do `git add main.c` followed by another `git status` and the file is now green under *files to be committed*.

Commit the file with `git commit -m "first file"` or something like that - you need double quotes if you want spaces in your commit message. Try `git status` again and you should see *nothing to commit, working tree clean* which means git is up to date with your files. Try `git log` and you will see that there is now one commit in the log.

Advanced note

Every git commit must have a commit message. You can either add one with the `-m` flag, or leave that off and git will drop you into the system default editor to write one. That is normally vim (the command to quit is press the escape key then `zz`). You can change the default editor by setting *environment variables* with command `export EDITOR=nano`.

If you want to keep this setting when you relaunch your shell next time you log in, then the `export` line has to go in a file called `.profile` in your home directory, which is a file that the bash shell processes when it starts up.

To keep a profile file around when vagrant rebuilds your VM you could stick a provisioning line in to your `vagrantfile` to ensure the `.profile` is updated:

```
echo 'export EDITOR=ed' >>~vagrant/.profile
```

Ignoring files

Compile your code with `gcc main.c -o program`, and check with `./program` that it runs and prints *Hi*. (If you get an error that `stdio.h` doesn't exist, then you have installed gcc but not the C development libraries Hint: `man apt-file .`)

If you look at `git status` now, the program file shows as untracked, but we do not want to commit it: the repository works best when you store only your source code, and anyone who needs to can check out a copy and build from there. Among other things this means that people on different platforms e.g. linux and mac, intel and ARM and so on can each compile the version that works for them.

So we want to tell git to ignore the program and changes in it, which we do by creating a file called `.gitignore` and adding an expression on each line to say which file(s) or folders to ignore - you can use `*.o` to select all object code files, for example.

- Create a file `.gitignore` and add the single line `program` to it.
- Do another `git status` and notice that while the program is now ignored, the ignore file is marked as new. This file does belong in the repository, so add it and commit it.

- Check that `git status` reports *clean* again, and that `git log` contains two commits.

Commit and checkout

As you develop, you should regularly code, commit, repeat. To practice this, change *Hi* to *Hello* in the program, rebuild and run the program, then add and commit the source file again - check with `git status` at the end that you get *clean* again.

The command `git add .` adds all new and changed files and folders in the current folder in one go, and is typically the quickest way to add things when you want to commit all your changes since the last commit.

Sometimes you want to go back and look at another commit, or undo a commit that broke something - this is when you want a checkout.

- Use `git log` to show the history of your commits. (When you have more than one screen, `git log |less` lets you scroll.)
- Note the first 6 or so characters of the commit hash of the commit where you added the ignore file, but before changing *Hi* to *Hello*. You need at least 6 characters, but only as many so that it's not ambiguous to git which commit you mean.
- Run `git checkout HASH` where HASH is the 6 or however many you need characters of the commit in question. Git will print a warning about the HEAD pointer.
- Check the source file, and notice that it is now back on *Hi*.
- Use `git checkout main` to return to the latest version of your files, and git will set up the HEAD pointer again ready to accept new commits.

Advanced note

If you actually want to undo a commit, then you have two options:

- `git revert HASH` adds a new commit that returns the files to the state they were before the commit with the given hash. This is safe to use during team development, as it's just adding a new commit. If you have commits A, B and do `git revert B` then you get a new commit C so anyone else using the repository sees a sequence of commits A, B, C; but the state of the files in C is the same as in A.
- `git reset HASH` undoes commits by moving the HEAD pointer back to the commit with the given hash, but leaves the working copy alone (you can use the `--hard` option to change the files as well). This will break things if you have shared your newer commits with other developers, but it's safe to use to undo changes that you haven't pushed yet (we'll learn about this next time). The effect is as if the commits which you've reset had never happened.

Note: if you want to revert a commit because you accidentally committed a file with secret information, and you've already pushed the commit, then you also have to look up online how to "force push" your changes to erase all traces of the file on github (or other online

providers). If the secret file contained any passwords, even if you reverted the commit immediately, then you should consider the passwords compromised and change them at once.

Part 2: Git forges

In this exercise we will set up and use a git forge account with a remote provider. The *typical* ones you usually see for hosting git repositories are:

- github.com
- gitlab.com
- bitbucket.org

But *many* more exist. You can even create your own with little more than an SSH server. Jo's favourite is one called [sourcehut](#) but you have to pay for it.

Advanced note

If you do want to build your own git server from scratch it isn't that hard but you have to muck about with *bare* git repos (that we don't cover) and set some funky file permissions. [Instructions can be found here for the brave](#).

This exercise is based on GitHub, as it is the most popular provider, but you can use one of the other two if you want as well---although the web user interface and some advanced features are different, interacting with all three on the command line is identical and all three offer unlimited private and public repositories (within reason).

Set things up

Go to github.com and register with a username, an e-mail address and a password. You might have to click a confirmation link in an e-mail sent to you.

We are going to use git over SSH, so you need to let git know your public key (remember, you never give anyone your private key!). Click the icon in the top right corner of the screen that represents your avatar (you can of course set a custom one if you like) and choose *Settings* in the menu, then on the settings page choose *SSH and GPG keys*.

Choose *New SSH key*, and paste your SSH public key in the `Key` box (you created one last week, see `man ssh-keygen`). Give your key a title if you like, then add it with the green button. Github supports all common SSH key formats, but will warn you if you do something silly like upload a private key or a key in an outdated and weak cipher. Some providers (Bitbucket) insist you use a specific type of key (usually `ed25519`): add the appropriate flag when generating the key to create it (`-t ed25519`) if you want that.

Advanced note

If you have many devices (desktop, laptop) that you work from and many servers (github, gitlab, lab machine etc.) that you connect to, how do you manage keys?

Whilst using the same public key for different services is not exactly a security problem: even if one service gets hacked and you connect to it while it's under the hacker's control, that does not leak your private key; it feels a bit icky. Generating keys is easy and it is barely any more work to have separate keys per server and per machine you use.

However, reusing public keys can be a privacy problem, because every service that you use the same public key (or e-mail address, or phone number etc.) can potentially work with the others to know that you are the same individual. It is no problem to use different keypairs for different services, in which case you probably want a block in your `~/.ssh/config` file with something like

```
Host github.com
  User git
  IdentityFile ~/.ssh/github.com.key.or.whatever.you.called_it.
```

Search the manual pages for `ssh_config` for full configuration options.

We are assuming that you will be running the git commands in the rest of this section on an VM, either on your machine or on a lab machine, however if you have git installed on your own machine directly (which is a good idea) then you can run this exercise there too.

A note on naming

The name of the main branch changes: it used to be called `master`. You may see the default branch named as either `master` or `main`, or something else entirely. So long as you are consistent, the name of the default branch doesn't matter at all (and you can configure it if you have a preference), and you just need to know that in these exercises we will use `main` to refer to the default branch and you should substitute that for your own default branch name if it is different.

Create a repository

On the main page, you should see an empty *Repositories* bar on the left, with a new button. Use that to create a repository, on the next page give it a name and tick the *Add a README file* box.

On the repository page, there is a green *Code* button. Clicking that opens a box with three tabs: *HTTPS*, *SSH* and *GitHub CLI*.

Each repository has a two-part name: the first part is the owner's github username, the second part is the repository name. For example, the repository for this unit is called `cs-uob/COMSM0085`. There are two ways to interact with a remote repository:

- Via HTTPS. This is ok if you are just cloning a public repository, as it does not require any authentication. To interact with a private repository or to push files, HTTPS requires username/password authentication, and we can do better than that.
- Via SSH, using keys. This is the recommended way to use Git.

Click the SSH tab and copy the URL there - it should be something like
`git@github.com:USERNAME/REPONAME.git`.

On the command line, run the command `git clone`

`git@github.com:USERNAME/REPONAME.git` where you replace USERNAME and REPONAME with the parts from the SSH tab of your repository. Git clones your repository and puts the content in a subfolder named after the repository name - you can change this by providing a different folder name as an extra command-line argument to `git clone`, or you can just move or rename the folder later on.

Note: certain OS/ISP/DNS combinations might get you "resource temporarily unavailable" when you try and access github via ssh. The problem is that the actual address is `ssh.github.com` and not all set-ups correctly pass on the redirection when you try and connect to github directly. If you are experiencing this error, you can either use `ssh.github.com` in place of `github.com`, or add an entry in your `~/.ssh/config` file as follows (if you have to create this file first, make sure it is not writable by anyone except yourself or ssh will refuse to accept it):

```
Host github.com
  Hostname ssh.github.com
  Port 22
```

Go to that folder, and try `git remote show origin`. Here, `origin` is the default name of a `remote`, and the result should look a bit like this:

```
* remote origin
  Fetch URL: git@github.com:USERNAME/REPONAME
  Push URL: git@github.com:USERNAME/REPONAME
  HEAD branch: main
  Remote branch:
    main tracked
  Local branch configured for 'git pull':
    main merges with remote main
  Local ref configured for 'git push':
    main pushes to main (up to date)
```

The bits about `main` are to do with branches, which we will discuss in another activity in more detail.

Advanced note

You can have several remotes with different names - for example if you fork (create your own copy of someone else's repository) then you get the original one as a second remote named *upstream*, so you can share changes back with them - this is the way you create new content for the [CSS website](#) for example.

You can also use folders as remotes: if you want to practice resolving merge conflicts, you could do the following:

```
mkdir remote
cd remote
git init --bare
cd ..
mkdir user1
git clone remote user1
mkdir user2
git clone remote user2
```

This gets you a remote and two "users" in different folders to play with. The remote was set up with `--bare` so that it does not contain a working copy, but acts as a pure repository.

You can now `cd user1` to simulate user 1 doing work, and can fetch/push/pull as described below. (Ignore warnings about "upstream", they will go away once you have committed a file to the repository.) Then you can `cd ../user2` to switch to a second working copy, which you can pretend is another user on another machine.

If you want to adjust the user names for the commits, then running `git config user.name "YOURNAME"` and `git config user.email "YOUREMAIL"` without the `--global` option from last time changes the settings just for one repository.

Do a `git status` and note that a new line appears compared to last activity:

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

This line comes in four versions:

- Up to date: there have been no commits on your local or the remote repository since you last synchronised.
- Ahead of remote: you have made commits locally that you have not yet pushed to the remote.
- Behind remote: someone else, or you on a different computer, have made commits to the remote that you do not have on this computer yet.
- Diverged from remote: both your computer and the remote have had different commits since the last time you synchronised.

Practice the push workflow

For this exercise, you should work in pairs or larger groups. Need someone to work with? Ask the person you're sitting next to.

One person creates a private repository (tick the box to add a README file) and adds everyone else in the group to it. You all need to have an account with the same provider for this to work.

- On Github, the way to add people to a repository is on the repository page: choose *Settings* in the top menu, then *Manage access*. Here you can press *Invite a collaborator* and enter their Github username. This causes Github to send them an e-mail with a link they need to click to accept the invitation and be added to the repository. *Note: you must be logged in to github when you click the link on the invitation e-mail, otherwise you will get an error message.*

Everyone `git clone`s the repository to their own Debian VM (or their own machine directly).

Everyone does the following, one person at a time doing all steps (coordinate among each other):

1. Imagine that it is mid-morning and you are starting on a day's coding.
2. First, make sure your terminal is in the folder with your working copy, and type `git fetch .`
 - If you get no update, then there were no changes on the remote since your last fetch and you are ready to start coding. (This should happen to the first person to do this step.)
 - If you get output, then there were changes on the remote. Do a `git status` to see details (everyone except the first person should see this). Notice the line `behind origin/main ... can be fast-forwarded.` which means you just need to `git pull` and you will get the latest version of the files. Do a `git log` too to see the last person's commit message.
3. Do some coding: make a change to the repository - add or change a file, then commit your changes. You can use `nano FILENAME` to create and edit a file in your terminal, if you have installed it as described in the last activity.
4. Run the following push workflow to push your changes to the remote:
 1. Do a `git fetch` to see if there were any remote changes (there shouldn't be, for this exercise).
 2. Do a `git status` and make sure you are `ahead of origin`, not diverged .
 3. Do a `git push` to send your changes to the remote.

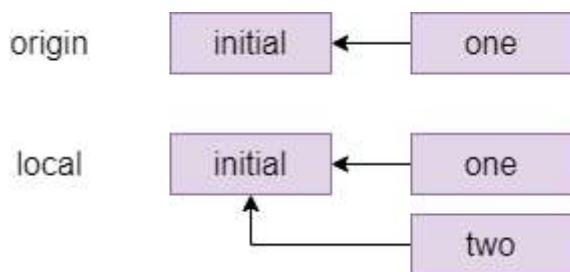
You can now code as a team, as long as only one person at a time is working - clearly not ideal.

Resolve a fake conflict, part one

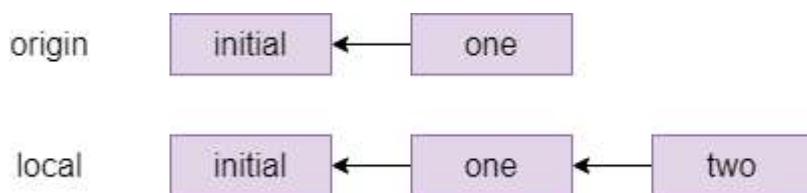
Produce a "fake" conflict as follows:

1. Two team members make sure they are up to date with their working copies (do a `git pull`, then `git status`). This represents you both starting coding in the morning.
2. One member adds or changes one file, then commits this change and pushes it by running the whole push workflow (fetch, status - check you're ahead, push).
3. At the same time as the first member is doing step 2, the second member adds or changes a different file, then commits this change. This represents two team members working in parallel, with the member one being the first one to complete their work and push a commit back to the remote.
4. The second member starts the push workflow with `git fetch`, then `git status`. Notice you have `diverged`. (If you were to try to `git push`, with or without fetching this would produce an error.)

The commit graph of member two looks something like this:



One way to resolve this conflict is a *rebase*, which is pretending that member two had actually fetched the `one` commit before starting their own work. The command for this which member two types is `git rebase origin/main` which means *pretend that everything in origin/main happened before I started my local changes* and gives the following graph:



Indeed, if member two does a `git status` after the rebase, they will see `ahead of origin/main by 1 commit` and they can now `git push` to send their local changes to the remote repository.

Different companies and teams have different opinions on when a rebase makes sense: some places forbid rebasing like this entirely, at least for work that is genuinely shared between different people. There is more or less a general consensus that you should not rebase when different people were editing the same files, but it is a technique worth knowing about for conflicts like the one you just created where different people have edited different files, as it makes for a cleaner commit graph.

Fake conflicts, part two

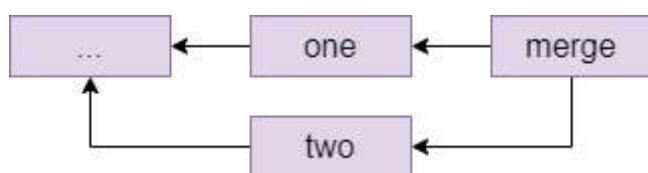
The other way to fix conflicts - and the only one that some people will use - is a merge. Let's do another fake conflict, but resolve it with a merge this time:

1. Two team members both get their repositories up to date with the remote. If you have just followed the instructions above then team member one has to `git pull` and team member two is already up to date because they have just pushed; both team members should check with `git fetch` then `git status` that they are `up to date`.
2. Like before, team member one edits one file, commits it and does the whole push workflow (`fetch`, `status` - check you're ahead, `push`). The second team member at the same time, without another `fetch`, edits a different file and commits.
3. The second team member starts the push workflow: `fetch`, `status` - notice you've diverged .

The second member's commit graph looks similar to the previous one before the rebase, perhaps with more commits in place of the *initial* one.

The second member is about to do a merge, which can either succeed (as it should here, because different people edited different files) or fail with a merge conflict (for example if different people edited the same file). If a merge succeeds, then git will make a merge commit and will drop them into their system's default editor, which is normally `vi`. Because we don't want to learn that right now, the second member should type `echo $EDITOR` in their shell and see what they get - if they get `nano` then they're fine, if they get an empty line then they should do `export EDITOR=nano`.

The second team member types `git pull`. Since this is a fake conflict (different files), this gets you into your editor, and you can see that on the first line is a suggested commit message starting with `Merge branch 'main'`, which it is conventional to accept without changes - exit your editor again. Git replies `Merge made by the recursive strategy.` and your commit graph now looks something like this (the `...` stands for the last commit from the previous section):



Resolving a real conflict

And next, we'll practice dealing with a real conflict, where two people have edited the same file.

1. Two team members get their repositories synchronised again: everyone does a `git pull`.
2. Team member one creates a file called `README.md` or edits it if it already exists, and adds a line like `Created by NAME` with their own name. Then they commit this change and run the push workflow: `git fetch`, `git status`, check they're ahead, `git push` to the remote.
3. Team member two, without fetching the latest commit, creates the same `README.md` file and adds a line `Created by NAME2` and commits this to their local repository. This simulates two people working on the same files in parallel since they both last fetched, and one of them (member one in this case) is the first to get their changes back to the remote.
4. Team member two starts the push workflow: `git fetch`, `git status` and notice that you have diverged again.
5. Run `git pull` as member two. You should see the following message:

```
CONFLICT (add/add): Merge conflict in README.md
Auto-merging README.md
Automatic merge failed; fix conflicts and then commit the result.
```

Open the file, for example `nano README.md` and notice that git has annotated it:

```
<<<<< HEAD
Created by NAME2.
=====
Created by NAME1.
>>>>> b955a75c7ca584ccf0c0bddccbcde46f445a7b30
```

The lines between `<<<<< HEAD` and `=====` are the local changes (team member two) and the ones from `=====` to `>>>>> ...` are the ones in the commit fetched from the remote, for which the commit id is shown.

Member two now has to resolve the conflict by editing the file to produce the version they would like to commit. For example, you could remove all the offending lines and replace them with `Created by NAME1` and `NAME2`.

Member two can now do the following:

- `git add README.md` (or whatever other files were affected).
- `git commit`. You could give a message directly, but a commit without a `-m` drops you into your editor and you'll see that git is suggesting `Merge branch main ...` as a default message here. It is conventional to leave this message as it is, just exit your editor without any changes.
- Run another push workflow: `git fetch`, `git status` and notice you are now ahead by 2 commits : the first one was the work you did, the second is the merge commit. You're ahead, so finish the workflow with `git push`.

Your commit graph looks the same as for the last merge that you did.

If you look at the repository's page on Github (<https://github.com/USERNAME/REPONAME>, where `USERNAME` is the name of the user who created the repository), then you can click on *Insights* in the top bar then *Network* on the left menu to see the commit history for the repository as a graph. Hovering over a commit node shows the committer, the message and the commit hash - and clicking on a node takes you to a page where you can see which changes were made in this commit.

On the main Github page for the repository, you can also click the clock icon with a number in the top right (on a wide enough screen it also shows the word *commits*) to go to a page showing all the commits on your repository in chronological order.

Working with others

In this activity you will practice Git the way it is used in real teams. You will need to form a group for this activity, ideally more than two students.

Set-up

One member makes a Git repository on one of the online providers, adds the other team members and shares the cloning URL. Everyone clones the repository.

The repository must have at least one commit for the following to work. This condition is satisfied if you chose your provider's option to create a readme file; if not then make a file now, commit it and push.

The develop branch

By default, your repository has one branch named `main`. But you don't want to do your work on this branch directly. Instead, one team member creates a `develop` branch with the command

```
git checkout -b develop
```

The team member who created the `develop` branch should now make a commit on it.

This branch currently exists only in their local repository, and if they try and push they would get a warning about this. What they need to do is

```
git push --set-upstream origin develop
```

This adds an "upstream" entry on the local develop branch to say that it is linked to the copy of your repository called `origin`, which is the default name for the one you cloned the repository from.

You can check this with `git remote show origin`, which should display among other things:

```
Remote branches:  
  develop tracked  
  main   tracked  
Local branches configured for 'git pull':  
  develop merges with remote develop  
  main   merges with remote main  
Local refs configured for 'git push':  
  develop pushes to develop (up to date)  
  main   pushes to main   (up to date)
```

Everyone else can now `git pull` and see the branch with `git branch -a`, the `-a` (all) option means include branches that only exist on the remote. They can switch to the develop branch with `git checkout develop`, which should show:

```
Branch 'develop' set up to track remote branch 'develop' from 'origin'.  
Switched to a new branch 'develop'
```

Feature branches

Every team member now independently tries the following:

- Make a new branch with `git checkout -b NAME`, choosing a unique name for their feature branch.
- Make a few commits on this branch.
- Push your feature branch with `git push --set-upstream origin NAME`.
- Make a few more commits.
- Do a simple `git push` since you've already linked your branch to `origin`.

Since everyone is working on a different branch, you will never get a conflict this way.

Anyone who is a project member can visit the github page can see all the feature branches there, but a normal `git branch` will not show other people's branches that you've never checked out yourself. Instead, you want to do `git branch -a` again that will show you all the branches, with names like `remotes/origin/NAME` for branches that so far only exist on the origin repository. You can check these out like any other branch to see their contents in your working copy.

Merging

When a feature is done, you want to merge it into develop. Everyone should try this, the procedure for this is

1. Commit all your changes and push.
2. Fetch the latest changes from origin (a simple `git fetch` does this).
3. `git checkout develop`, which switches you to the develop branch (the changes for your latest feature will disappear in the working copy, but they're still in the repository). You always merge into the currently active branch, so you need to be on `develop` to merge into it.
4. `git status` to see if someone else has updated develop since you started your feature. If so, then `git pull` (you will be *behind* rather than *diverged* because you have not changed develop yourself yet).
5. `git merge NAME` with the name of your feature branch.
6. Resolve conflicts, if necessary (see below).
7. `git push` to share your new feature with the rest of the team.

If no-one else has changed `develop` since you started your branch, or if you have only changed files that no-one else has changed, then the merge might succeed on the first attempt. It's still a good idea to check that the project is in a good state (for example, compile it again) just in case, and fix anything that's broken on the develop branch.

If the merge fails with a conflict, then you need to manually edit all the conflicted files (`git` will tell you which ones these are, do `git status` if you need a reminder) and `git commit` again.

The workflow for merging and resolving conflicts is essentially the same as the one from the last session, but since everyone is developing on a separate branch, the only time when you have to deal with a possible merge conflict is when you are merging your changes into `develop` - your own branches are "private" and you don't have to worry about hitting a conflict if you quickly want to commit and push your changes as the last thing you do before going home at the end of a long day's work.

Pull requests

Pull requests are not a feature of the git software itself, but of the online providers. They let a team discuss and review a commit before merging it into a shared branch such as `develop` or `main`. Depending on the provider, branches can also be protected or assigned owners so that only the branch owner or developers with the right permissions can commit on certain branches.

The procedure for merging with a pull request on github, which you should try out:

- Commit and push your feature branch.
- On [github.com](#) in your repository, choose *Pull Requests* in the top bar, then *New Pull Request*.
- Set the *base* branch as the one you want to merge into, e.g. `develop`, and the *compare* branch as the one with your changes. Select *Create Pull Request*.
- Add a title and description to start a discussion, then press *Create Pull Request* again to create the request.

Anyone in the team can now go to *Pull Requests* in the top bar of the repository page and see the open requests. You can either comment on them, or if it is your role in the team to approve the request for this branch, you can approve the pull request which creates a merge.

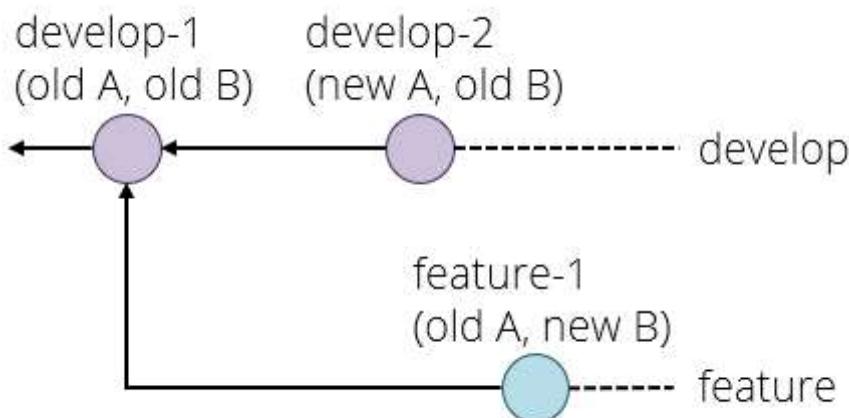
Since a pull request is linked to a branch, you can use it for code review as follows:

1. A developer creates a feature branch and submits a pull request.
2. The reviewer looks at the request. If they find bugs or other problems, they add a comment to the discussion.
3. The developer can address reviewer comments by making a new commit on their feature branch and pushing it, which automatically gets added to the discussion.
4. When the reviewer is happy, they approve the request which merges the latest version of the feature branch into the base branch (for example `develop`).

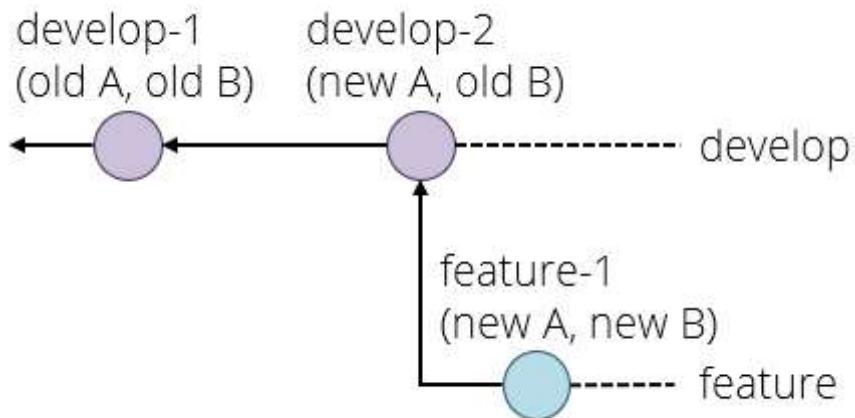
There is just one complication left. Suppose the following happens:

- Your project starts out with commit `develop-1` setting up the initial version of the `develop` branch. Imagine there are two files, A and B.
- You create a feature branch and make a commit `feature-1` which changes only file B.
- In the meantime, someone else makes a feature that changes file A, and merges it as `develop-2` to the `develop` branch.

You are now in the situation that `develop-2` has (new A, old B) and your `feature-1` has (old A, new B). Neither of these is what you want, you presumably want (new A, new B). We have met this situation before, but without branches. Graphically:



The solution here is to *rebase* your branch onto the latest commit on develop with `git rebase develop` and fix any conflicts that that causes, which produces the following situation:



If you now try and push your feature branch, you might get an error because the version of your feature branch on the origin repository still has the old version. The solution here is to force the push, which overwrites the old version, with

```
git push --force origin BRANCHNAME
```

This is a *think before you type* kind of command because it can break things for other developers if you do it on a shared branch. The basic safety rules are:

- Only rebase on *private* branches.
- Only force push on *private* branches, and only if it is absolutely necessary (for example to tidy up a rebase).

A private branch is one that you know no-one else is working on, for example your own feature branches.

If you ever get into a situation where you need to rebase or force push on a shared branch such as develop or main, you generally need to make sure that everyone on the team knows what is going on, synchronises their repositories both before and after the dangerous operation, and does not make any commits or pushes while someone is working on it - basically they need, in concurrency terms, an exclusive lock on the whole repository while doing this operation.

This is one reason why the main and develop branches are kept separate - and some workflows even include a third branch called `release`. If merges into main or release only ever come from the develop branch, then a situation where you need to rebase these branches can never happen.

To summarise, the pull request workflow is:

1. Commit and push your changes.
2. If necessary, rebase your feature branch on the develop branch.

3. Create a pull request.
4. If necessary, participate in a discussion or review and make extra commits to address points that other developers have raised.
5. Someone - usually not the developer who created the pull request - approves it, creating a merge commit in develop (or main).