

UNIX WORKSHOP

Dimitri Robl



CC BY-NC 2.5 by Rendall Monroe

<https://creativecommons.org/licenses/by-nc/2.5/legalcode>

This work is licensed under the [Creative Commons Attribution-Non-Commercial 4.0 International License](#)

Welcome to this three-day workshop which will teach you the basics of UNIX, (GNU/)Linux, and the command-line! There is a lot to learn ahead of us, but first, you need to set up your working environment.

Contents

1	Installing a Virtual Machine	5
1.1	Prerequisites	5
1.1.1	Enable Virtualization	5
1.1.2	Install Oracle VirtualBox	5
1.1.3	Getting the Installation Image	6
1.2	Installing a Virtual Machine Running Debian GNU/Linux	6
2	UNIX, GNU, and Linux	12
2.1	UNIX	12
2.2	GNU	13
2.3	Linux	13
3	Basics	14
3.1	Shells, Terminals, and Commands	14
3.2	The Filesystem	18
3.3	Help Utilities	20
3.3.1	man	20
3.3.2	info	21
3.3.3	--help and -h	22
3.3.4	help and type	22
3.3.5	which and whereis	23
3.3.6	what is	23
3.3.7	The Internet Is Your Friend	24
4	The Command-Line	27
4.1	Moving and Looking Around	28
4.2	Working with Files	31
4.3	Working with Content	39
4.3.1	Looking at Content	39
4.3.2	Creating Content and I/O redirection	41
4.4	Archiving and Compression	45
4.4.1	Archiving with tar	45
4.4.2	Compression Utilities	47
4.4.3	Combining Archiving and Compressing Data	49
4.5	Textfiles, Pipes, Wildcards, and a Little Bit of Regex	50
4.5.1	Of Heads and Tails	50
4.5.2	Sorting Text and Using Pipes	52

4.5.3	Wild Cards and Gentle Regexes	55
4.5.4	Cutting and Joining	63
4.5.5	Editors (a.k.a. Religions)	67
4.6	Processes	68
4.6.1	Process Control	70
4.7	Variables and Your Environment	71
4.7.1	Variables	72
4.7.2	Your Environment	73
4.8	Shell Scripting Basics	76
4.8.1	Syntax of Common Constructs	80
4.9	Installing Programs With and Without Rootly Powers	81
4.9.1	Package Management	81
4.9.2	Installing Programs from Source	82
4.9.3	Your Very First Compiled Program	83
5	Final Words	85

1 Installing a Virtual Machine

In order to make life easier for you, it won't be necessary that you install a Linux distribution or distro¹ on your harddrive. Instead, we will guide you through the installation of a virtual machine² running Debian GNU/Linux. If you already have a working installation of any Linux distro (either physical or virtual) you can skip this step.

1.1 Prerequisites

In order to use a virtual machine (VM), your physical machine should have at least two cores, 4GiB of main memory (i.e. RAM) and there should be at least 20GB free space on your disk. In this case more is always better.

1.1.1 Enable Virtualization

The first thing to check is whether your CPU supports virtualization, and has it activated. Follow these steps:

1. Boot your computer.
2. Depending on your motherboard press ESC, DEL, ENTER, F2, F10, or F12.
3. Look for an option which says "Advanced" or "CPU settings".
4. Depending on whether you use an Intel or an AMD CPU you will have to enable Intel Virtualization Technology, Intel VT, VT-x, AMD-V, AMD Virtualization or similar.
5. Check if this option is enabled and if not, enable it.
6. Save changes and Exit.

1.1.2 Install Oracle VirtualBox

Now you have to install the software to run virtual machines:

1. Download the installer from <https://www.virtualbox.org/>. It runs on Windows, OS X, Linux and Solaris hosts.
2. Start the installer and follow the instructions.

¹You'll learn what exactly that means later in the course.

²If you are unfamiliar with this concept you can read the [Wikipedia entry](#) about it.

1.1.3 Getting the Installation Image

The final thing to obtain before the installation can begin is an image to install. Just download the **netinst Debian image** and store it somewhere you remember(!), you'll need it in a moment.

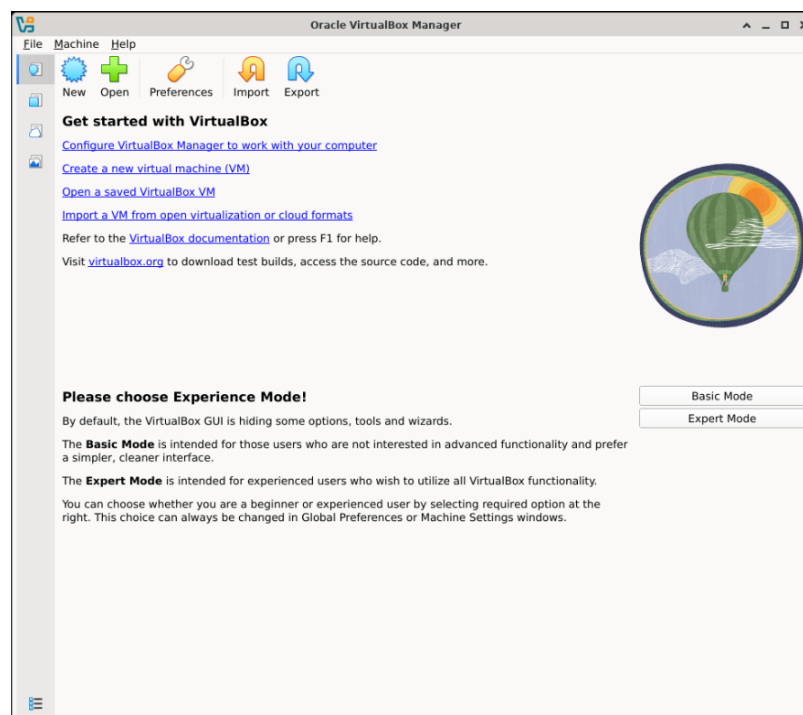
In case you don't have an x86_64 architecture ³, installation files for other architectures are available at <https://www.debian.org/distrib/netinst>. However, it is rather unlikely that you have a different architecture on a general purpose computer, unless you are still on a machine running 32-bit x86 ⁴ or own one manufactured by Apple before 2006 which has a PowerPC CPU or after 2020 when they switched to their own CPUs.

Now everything is ready to proceed to the installation!

1.2 Installing a Virtual Machine Running Debian GNU/Linux

To install a VM running Debian GNU/Linux follow the instructions below:

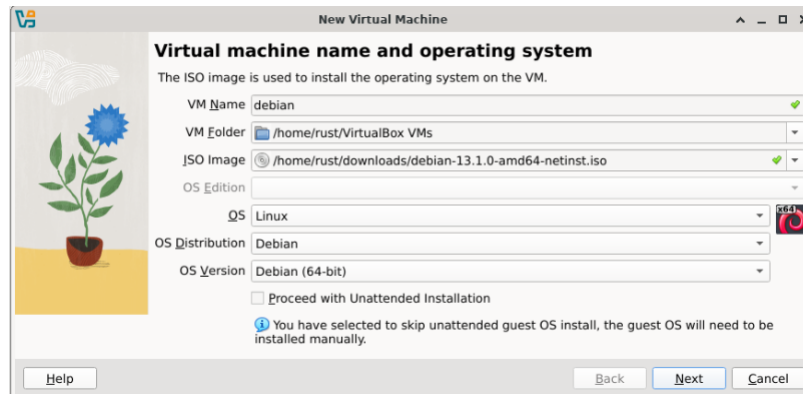
1. Open VirtualBox.
2. Click on the “New” button.



³To make things less confusing, this architecture is also called x64, AMD64, and Intel 64.

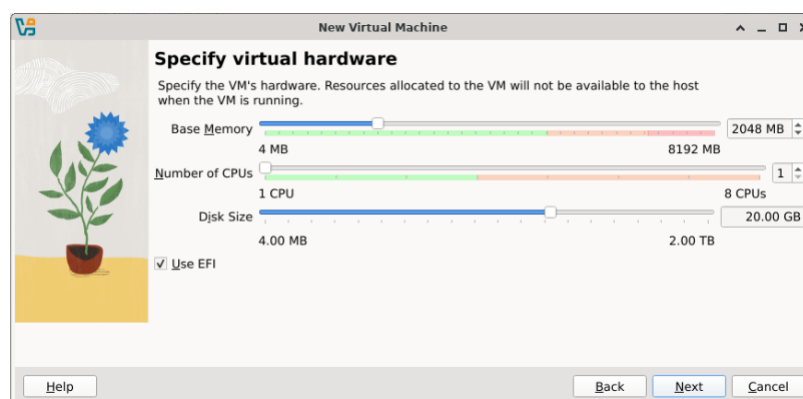
⁴A.k.a. IA-32, i386.

3. Enter a name for your VM, the directory VMs should be stored in, and the ISO-image you want to install the machine from. If this is not automatically done, select *OS* → *Linux*, *OS Distribution* → *Debian*, and *OS Version* → *Debian (64-bit)*.⁵ Press “Next”.



4. Select the amount of memory the VM will have. The best value here depends on what you want to do with the VM and how much RAM your physical machine has. The default value should be sufficient for our purposes, but you can use more if you want and less if you are very low on physical RAM (i.e. 2GiB or less).

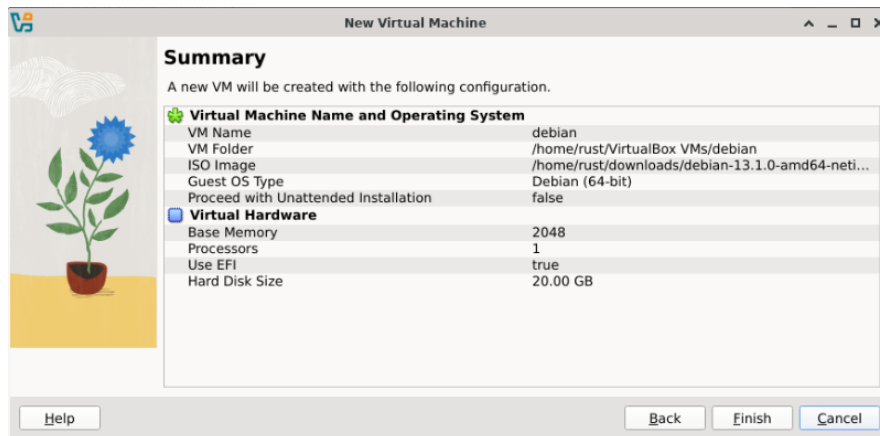
Also choose the number of CPUs you want to allocate to the VM and how much disk space to give it. Again, the default values are fine for our purposes.



You can select “Use EFI” as in the image but you don’t have to.

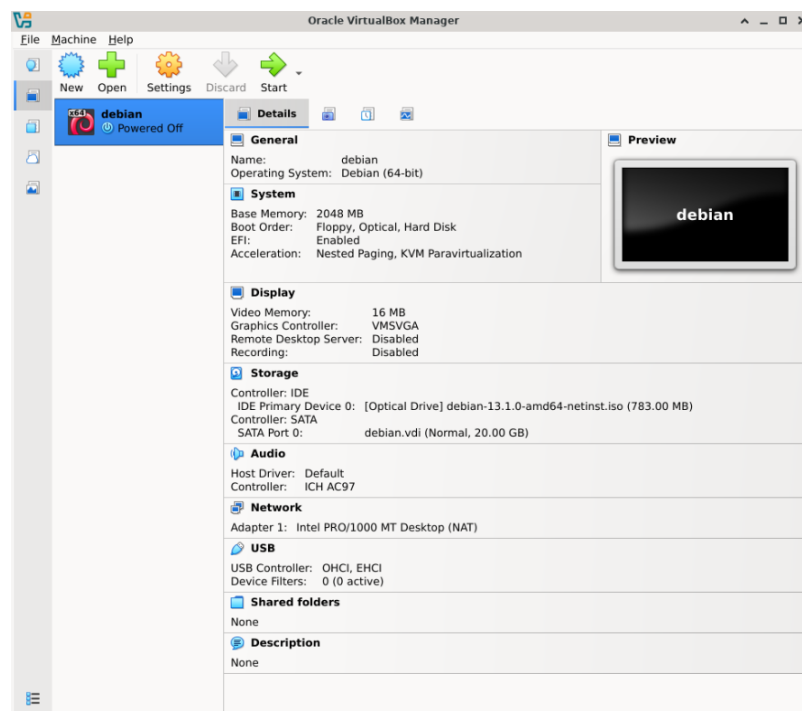
5. Next, you’re presented with an overview of your soon-to-be VM which should look something like this:

⁵If you use a different architecture than x86_64 you have to specify this here as well.



Click “Finish”.

6. Now you’re back to the main screen, which should look something like this:

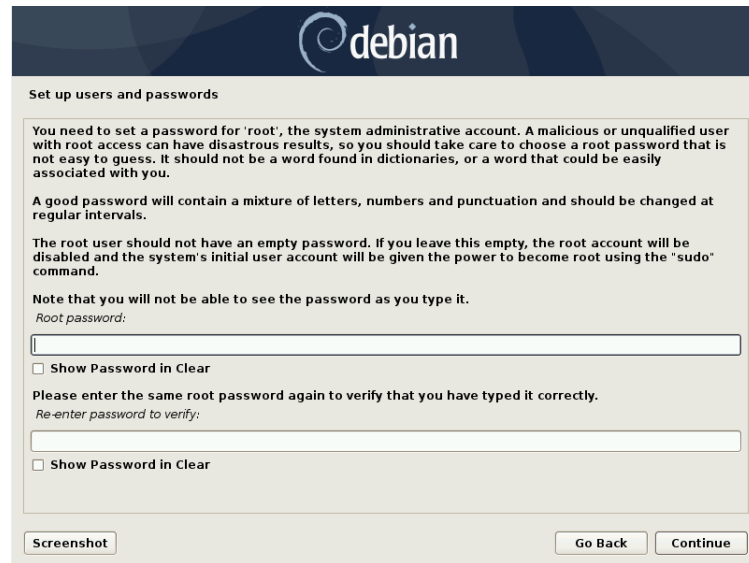


Select you newly created VM and press “Start”.

That’s it, a new window should pop up and the installation of your new virtual OS can begin. Select “Graphical Install” and follow the on-screen instructions. There are some things you should be careful about:

1. The selection of your locale (i.e. the language the system will use) and your keyboard layout – e.g. if you want to use an English system, but a German keyboard you have to tell the installer.

2. When asked to enter the root password, leave it blank to disable the root account and use `sudo` instead. We'll discuss the details later.



The image shows a Debian installer window titled "Set up users and passwords". It contains instructions on setting a root password, a warning about password strength, and a note that the root account will be disabled if the password is left empty. There are two password input fields, each with a "Show Password in Clear" checkbox. At the bottom, there are "Screenshot", "Go Back", and "Continue" buttons.

debian

Set up users and passwords

You need to set a password for 'root', the system administrative account. A malicious or unqualified user with root access can have disastrous results, so you should take care to choose a root password that is not easy to guess. It should not be a word found in dictionaries, or a word that could be easily associated with you.

A good password will contain a mixture of letters, numbers and punctuation and should be changed at regular intervals.

The root user should not have an empty password. If you leave this empty, the root account will be disabled and the system's initial user account will be given the power to become root using the "sudo" command.

Note that you will not be able to see the password as you type it.

Root password:

☐ Show Password in Clear

Please enter the same root password again to verify that you have typed it correctly.

Re-enter password to verify:

☐ Show Password in Clear

Screenshot Go Back Continue

3. If you don't know about disk partitioning leave the defaults intact.



The image shows a Debian installer window titled "Partition disks". It explains the difference between guided and manual partitioning. Under the "Partitioning method:" section, there is a list of options: "Guided - use entire disk", "Guided - use entire disk and set up LVM", "Guided - use entire disk and set up encrypted LVM", and "Manual". At the bottom, there are "Screenshot", "Go Back", and "Continue" buttons.

debian

Partition disks

The installer can guide you through partitioning a disk (using different standard schemes) or, if you prefer, you can do it manually. With guided partitioning you will still have a chance later to review and customise the results.

If you choose guided partitioning for an entire disk, you will next be asked which disk should be used.

Partitioning method:

Guided - use entire disk

Guided - use entire disk and set up LVM

Guided - use entire disk and set up encrypted LVM

Manual

Screenshot Go Back Continue

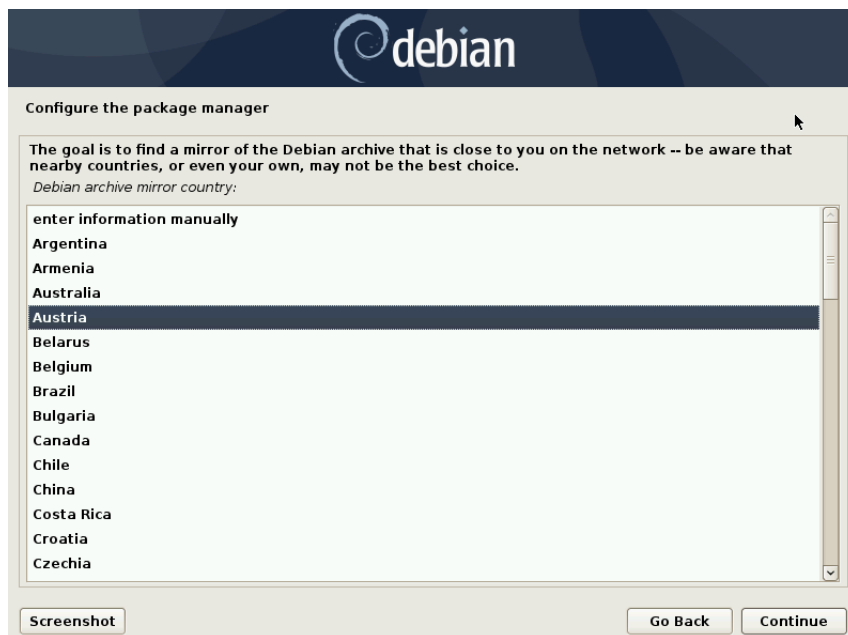
However, in case you install Debian GNU/Linux on real hardware (especially on a portable device such as a laptop) and not as a VM, we *strongly* recommend using the "Guided - use entire disk and set up encrypted LVM" option. This will encrypt your harddrive which means that every time your PC or laptop starts you will have

to enter a passphrase to decrypt the disk before the OS even boots. This has the huge advantage that in case your device gets stolen, it will be almost impossible to access your data if the thief does not your password/passphrase – and no one should ever know that! Make sure to choose a strong password, where “strong” means “as long as possible”. You don’t necessarily have to care about special characters, length is far more important. See [here](#).

The only downside to this is the fact that if you forget or lose your password/-passphrase you yourself also won’t be able to access your data. This is one of the few cases in which it *may* be a good idea to write down your password/passphrase on a piece of paper and store it in some secure place.

Also, don’t worry about the warnings when it says “Write changes to disk” if you set up a VM. The VM ‘believes’ it has a whole disk to write on and is not aware that it is actually just a file in the underlying OS. If you followed the instructions so far, it is not possible to destroy your real hard disk by installing the VM.

4. When you come to select a mirror to download the packages, double-click on “Austria”



Then select any mirror. As it says in the description `deb.debian.org` is usually a good choice.



- Once you come to the screen “Software selection” we recommend selecting “Xfce” or “LXDE” if the VM is on low memory (< 2GiB), as these are very lightweight GUIs. Otherwise you can stick with the default “Debian desktop environment”.



If you run into any problems, use your favorite search engine and look for something like “Debian installation tutorial”. Most likely you’ll find written instructions as well as videos on how to do it.

2 UNIX, GNU, and Linux

This section seeks to clarify terms you are very likely to encounter pretty soon when working with any (GNU/Linux) distro. Be prepared that people are *very* picky about words, and don't get discouraged in case you meet a particularly nasty partisan of one of these wordisms.

2.1 UNIX

In the beginning there was ... the desire to play games! Yes, that's it! In the 1960ies two MIT students called Dennis Ritchie and Ken Thompson wanted to play the game "Space Travel" on their new, but underpowered minicomputer¹ PDP-7. To achieve their goal, they had to tweak the machine a bit and as they were researching operating systems (OS) at the time anyway, they ended up writing a whole new OS for the PDP-7 which came to be named UNIX.

UNIX, based on the even older **Multics**, introduced a lot of new concepts into OS-design, many of which are still in use today. UNIX was developed by Bell Labs which where prohibited from selling it under US-law, but in 1983 this decree was lifted and AT&T, the owner of Bell Labs promptly tried to turn UNIX into a product, as it was already widely used among computer science students and companies they worked for. However, by this time there were already multiple versions of UNIX around, and one of the most important of them was developed by researchers in Berkeley; the Berkeley Software Distribution (BSD).

BSD was sued by AT&T for copyright infringement despite the fact that large parts of AT&T's own UNIX implementation was based on code from Berkeley. After a long legal battle, BSD won its freedom and the base of modern FreeBSD, OpenBSD, NetBSD, etc. was set.

UNIX finally lost the battle for the PC against Apple and Microsoft, which was very likely at least in parts based on the law suits.

However, in the server market and in technical and scientific circles, UNIX and its derivatives continued to be state of the art and remain so today. Netflix' and WhatsApp's infrastructure, for example, run on FreeBSD. And Sony's Playstation 4 also runs a customized version of FreeBSD.

¹"Mini" here indicates that it didn't fill a whole room.

2.2 GNU

GNU (GNU's Not Unix) was founded in 1983 by Richard M. Stallman after he was fed up with the closed source philosophy of UNIX. He wanted to create an OS that is “free as in freedom”, meaning it adheres to the four principles of free software, which state that users have to be

0. able to run the program as they wish, for any purpose.
1. able to study how the program works and can change it according to their wishes.
2. able to redistribute copies to others, either free or for a fee.
3. able to distribute modified versions to others, again either for free or for a fee.

Many tools necessary for a complete OS were created by the GNU project, including gcc (GNU C Compiler), g++ (GNU C + + Compiler), gdb (GNU Debugger), a shell, and others.

However, one essential part was incomplete: A working kernel. The kernel is the part of the OS which interacts directly with the hardware, so without a kernel, an OS can't run. Thus, the GNU tools were used on a number of UNIX systems, but a stand-alone OS was still wanting.

This changed when in 1991 a Finnish computer science student by the name of Linus Torvalds released a kernel he (eventually) called “Linux”.

2.3 Linux

Linus Torvalds originally released the Linux kernel under a license he created himself, but in 1992 he re-licensed it under the GPLv2 (General Public License, version 2) which is a license of the GNU project.

Technology-enthusiasts and academia quickly started using Linux as their kernel and as soon as it was GPL-ed there existed a full GNU OS with a working kernel. This is why you may read/hear the term “GNU/Linux”. Be careful here, some people are *very* picky about using “Linux” only when referring to the kernel and “GNU/Linux” when referring to an operating system using a lot of GNU-tools and running Linux as a Kernel.

Linux' open source nature made it possible that its development proceeded very quickly with thousands of people contributing code and time. The current major release of Linux is version 5, the original 5.0 kernel was released in 2019. While Linux is often hailed as *the* example of a successful open source project with loads of people sacrificing their spare time to make the Linux kernel better, this is largely obsolete today. For the kernel releases between 4.8–4.13, 87.7% to 91.8% of the changes have been done by people who are paid for this by various companies.² This is not necessarily a bad thing, it just shows that the IT-industry heavily relies on Linux and therefore actively develops it and also that Linux today is very far from being a hobbyists' project but a mature OS kernel developed mostly by professionals.

²See [here](#). At the time of this writing no report on 2018 was available.

3 Basics

This section will introduce important concepts and words you'll have to work with, show you how to help yourself on a (GNU/Linux) system in case you get stuck and explain the filesystem.

To follow along, please use the provided .tar-file.

On the command-line you can type (or just copy-paste – which can be done by just selecting text with your mouse to copy it and paste it by pressing the middle mouse button on most modern GUIs on (GNU/Linux) the following to unpack the archive containing the folder:

```
tar -xvzf unixcourse.tgz
```

3.1 Shells, Terminals, and Commands

In the last chapter you've seen the concept of the kernel, the part of the OS which talks to the hardware. If you want to communicate with the kernel you need something, that wraps around it – a *shell*, for example.

Essentially, a shell is a user interface which allows you to access the services provided by the OS. You are most likely familiar with a graphical user interface (GUI) which allows you to access some services of the OS using a mouse and icons. A shell, on the other hand, requires text as input by providing you with a *command line*. While the GUI is often part of the OS you buy, i.e. you can't easily change the program that is your GUI if you use Microsoft Windows or Apple's OS X, a shell on a (GNU/Linux) system is just a program and you can choose from a wide variety of options. The shell we are going to work with here is the **GNU bash** (Bourne Again SHell), which is based on an older shell written by Stephen Bourne in 1979 and is obviously also a wordplay on the verb 'born'. We chose this one because it is currently the standard shell on all major (GNU/Linux) distros, but do play around with others if you like to.

The stuff you enter on the command-line to your shell are *commands* and they work the same way as clicking an icon with your mouse does, e.g. you can either double-click on the Firefox icon, or just type the word "firefox" on the command line, followed by ENTER. Both actions will start a new instance of the Firefox webbrowser (if it is installed). One thing to keep in mind in the following sections is that most commands are actually stand-alone programs, even though they write their output to the terminal screen, while only a few of them are so-called *shell builtins*, i.e. keywords the shell itself understands and works on without starting a new program. This will be of special importance in 5 where we talk about utilities that provide documentation for commands.

Many commands can take arguments and options. Their basic syntax is usually:

command [OPTION]... [ARGUMENT]...

The brackets around OPTION and ARGUMENT indicate that they are optional and the dots tell you that the preceding entity can be used multiple times, i.e. the line reads approximately as “Use command by typing command, possibly followed by one or more option(s), possibly followed by one or more argument(s).” Arguments tell the command e.g. on which file to operate, whereas options tell it what to do with the arguments. To separate these things, you have to put a space after the command, after each argument, and after each (group) of options.

Options come in two flavors: Short and long ones. Short options are usually indicated by a single, non-separated, preceding dash ‘-’, while long options can be recognized by two non-separated preceding dashes ‘--’. A very simple example would be

```
$ firefox --new-window https://www.univie.ac.at
```

which, little surprisingly opens the homepage of the University of Vienna in a new window of the Firefox webbrowser.

In most cases options for a single dash consist of a single letter, while options preceded by two dashes are words or phrases, e.g. let’s say you use the `ls` program (to *list* directories and/or files) which has the options `-l`, `-a`, and `--human-readable` and want it to process argument `arg`. To do so, you could type:

```
$ ls -a -l --human-readable unixcourse
total 524K
drwxr-xr-x  9 me me 4.0K Sep 24 22:43 .
drwxr-xr-x  4 me me 4.0K Sep 24 22:45 ..
drwxr-xr-x 13 me me 4.0K Sep 19 10:05 contentdir
drwxr-xr-x  2 me me 4.0K Sep 18 15:36 dir1
drwxr-xr-x  2 me me 4.0K Sep 17 13:52 dir2
drwxr-xr-x  2 me me 4.0K Sep 17 13:52 dir3
drwxr-xr-x  2 me me 4.0K Sep 17 15:36 'dir with spaces'
-rw-r--r--  1 me me   63 Sep 19 11:07 errorfile
-rw-r--r--  1 me me   29 Sep 18 15:24 file1
-rw-r--r--  1 me me   15 Sep 23 12:32 file10
-rw-r--r--  1 me me   11 Sep 23 12:22 file3
-rw-r--r--  1 me me    0 Sep 24 22:43 filetodelete
-rw-r--r--  1 me me  472 Sep 23 09:49 grepfile
-rw-r--r--  1 me me    0 Sep 23 12:33 hasfileinit
-rw-r--r--  1 me me 422K Sep 18 17:52 hugefile
-rw-r--r--  1 me me   46 Sep 23 12:11 join1
-rw-r--r--  1 me me   48 Sep 23 12:11 join2
-rw-r--r--  1 me me  405 Sep 23 12:22 linefile
-rw-r--r--  1 me me  1.4K Sep 23 12:29 lsfile
-rw-r--r--  1 me me   26 Sep 23 12:22 numeric sort
-rw-r--r--  1 me me   25 Sep 17 13:53 onelinefile
```

```

-rw-r--r-- 1 me me 69 Sep 18 15:36 renamedfile
drwxr-xr-x 2 me me 4.0K Sep 18 11:14 rmdirectory
drwxr-xr-x 2 me me 4.0K Sep 18 15:18 rminteractive
-rw-r--r-- 1 me me 331 Sep 23 11:02 sedfile
-rw-r--r-- 1 me me 331 Sep 23 10:48 sedfile.bak
-rw-r--r-- 1 me me 18 Sep 23 12:22 sortfile
-rw-r--r-- 1 me me 747 Sep 23 11:25 tablefile

```

One thing to remember is that it is possible to combine short options behind a single dash, i.e. the following would be equivalent to the above:

```

$ ls -al --human-readable unixcourse
total 524K
drwxr-xr-x 9 me me 4.0K Sep 24 22:43 .
drwxr-xr-x 4 me me 4.0K Sep 24 22:45 ..
drwxr-xr-x 13 me me 4.0K Sep 19 10:05 contentdir
drwxr-xr-x 2 me me 4.0K Sep 18 15:36 dir1
drwxr-xr-x 2 me me 4.0K Sep 17 13:52 dir2
drwxr-xr-x 2 me me 4.0K Sep 17 13:52 dir3
drwxr-xr-x 2 me me 4.0K Sep 17 15:36 'dir with spaces'
-rw-r--r-- 1 me me 63 Sep 19 11:07 errorfile
-rw-r--r-- 1 me me 29 Sep 18 15:24 file1
-rw-r--r-- 1 me me 15 Sep 23 12:32 file10
-rw-r--r-- 1 me me 11 Sep 23 12:22 file3
-rw-r--r-- 1 me me 0 Sep 24 22:43 filetodelete
-rw-r--r-- 1 me me 472 Sep 23 09:49 grepfile
-rw-r--r-- 1 me me 0 Sep 23 12:33 hasfileinit
-rw-r--r-- 1 me me 422K Sep 18 17:52 hugefile
-rw-r--r-- 1 me me 46 Sep 23 12:11 join1
-rw-r--r-- 1 me me 48 Sep 23 12:11 join2
-rw-r--r-- 1 me me 405 Sep 23 12:22 linefile
-rw-r--r-- 1 me me 1.4K Sep 23 12:29 lsfile
-rw-r--r-- 1 me me 26 Sep 23 12:22 numericssort
-rw-r--r-- 1 me me 25 Sep 17 13:53 onelinefile
-rw-r--r-- 1 me me 69 Sep 18 15:36 renamedfile
drwxr-xr-x 2 me me 4.0K Sep 18 11:14 rmdirectory
drwxr-xr-x 2 me me 4.0K Sep 18 15:18 rminteractive
-rw-r--r-- 1 me me 331 Sep 23 11:02 sedfile
-rw-r--r-- 1 me me 331 Sep 23 10:48 sedfile.bak
-rw-r--r-- 1 me me 18 Sep 23 12:22 sortfile
-rw-r--r-- 1 me me 747 Sep 23 11:25 tablefile

```

Long options cannot be combined, so you have to type each of them in full. Many programs have short and long options for the same operations and it's just a question of taste which one you use. A good approach is to start using long options because it is

easier to remember what they do and start memorizing the short options for operations you use a lot.

But why bother to learn the names of programs if you can just click on an icon? Surely, that's much more intuitive and simple! Well, it depends. Imagine you commonly use about 50 programs, many of which interact with each other – a screen with 50 icons for programs seems rather overwhelming, don't you think? And how about moving stuff from one window to the other all the time – annoying, I'd say. Or opening the same 7 programs one after the other, hundreds of times and copying data from one to the other? Convenience is clearly not the word for this.

These are some of the tasks that can be solved easily on the command line. Most command-line utilities follow the UNIX philosophy that each program should solve a single problem and solve it well. By combining the capabilities of many programs you get a very flexible toolset to solve your problem at hand. Make the easy things easy, and the hard things possible!



Figure 3.1: A DEC VT100 terminal. CC BY-SA [ClickRick](#)

Finally, a *terminal* is actually a piece of hardware (see 3.1), which can take input and display output. In this course and most likely in your whole life, you'll be confronted with terminal emulators, i.e. software which behaves like a terminal, like the one you see in 3.2.

When you open a terminal emulator your shell will be started and will provide you with a *prompt*, which usually looks a bit like

```
user@machine ~$
```

where *user* is your username, *machine* is the name of your computer and *~* is a shorthand for your home directory (see below for an explanation).¹ All your shell does now, is to wait for you to issue a command – let's do it the favor: type some random stuff and press ENTER. It should look something like this:

```
$ ajsdggasd
bash: ajsdggasd: command not found
$
```

¹From now on, we will abbreviate your prompt as just '\$'.

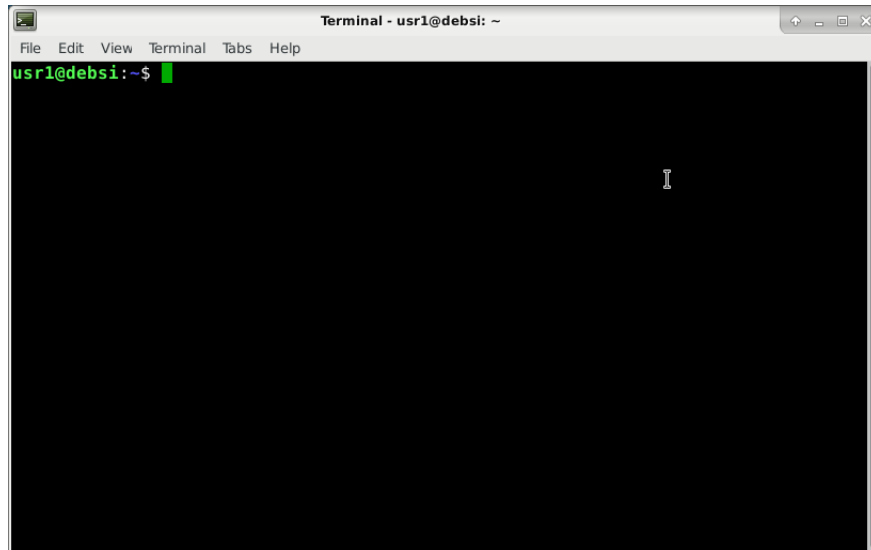


Figure 3.2: XFCE's terminal emulator

Little surprisingly, this gibberish does not constitute a command. But fear not, we shall use existing commands starting in the next section!

3.2 The Filesystem

In order to use the filesystem efficiently it is essential to understand its basic structure and this is what this section wants to teach you.

The filesystem is software which manages how your files and directories are stored on you disk/pendrive/CD/etc. Usually there are at least three kinds of entities in a filesystem:

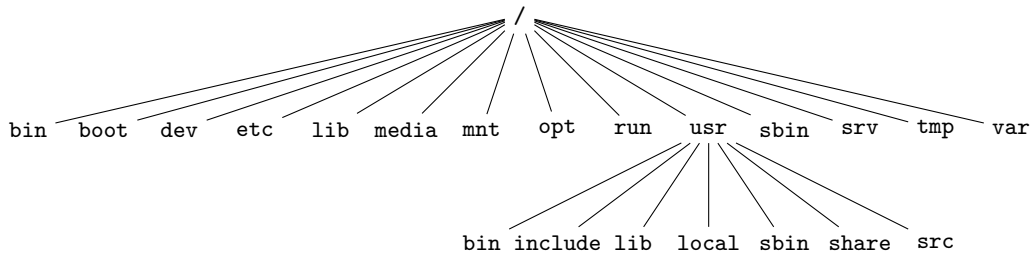
Files Some kind of data, possibly in a special format (e.g. PDF).

Directories These are places in which files are stored.

Links Links are references to either a file or a directory, i.e. they don't contain data, but just point to another place in the filesystem.

Filesystems in any UNIX system are usually structured as rooted trees.² The root of the filesystem is designated by the slash '/'.

²For the mathematically interested: A rooted tree is an acyclic undirected graph in which one vertex has been designated the root.



As you are logged in and opened a terminal already (if not, do so now), you can type

```
$ pwd
/home/me
```

Of course, you'll get a slightly different output, but there are two important things here:

1. `pwd` is our first real command! It stands for 'print working directory' which prints you current working directory, i.e. tells you where in the filesystem you are.
2. The output, `/home/me` tells you, that you are in a directory `me` which is in a directory called `home`, which is in the root directory `/`.

Okay, now that you know where you are, you may want to know which files³ are in this directory – you can list them using

```
$ ls
Documents Downloads Pictures Private
```

Again your output will differ slightly. So now, you used the program `ls` to show you the contents of the directory you are in. This is `ls`' standard behavior: If you don't provide any arguments it will display the content of you current working directory.

However, you can tell `ls` which directory it should list by providing a *path* as an argument. To show the contents of the root directory, type:

```
$ ls /
bin boot dev etc lib media mnt opt run sbin srv tmp usr var
```

The output provided here shows everything that is required in the root directory according to the [Filesystem Hierarchy Standard \(FHS\)](#), a document which defines what has to be in the root directory of any (GNU/)Linux distro.⁴ But as it only defines the minimum, distros are free to add other things and you'll most likely also find at least `home`, `proc`, and `sys` in your root directory.

Now there is one more concept you have to understand regarding the filesystem: *absolute* and *relative* paths. In a rooted tree you can specify any point in the tree by starting at the root and listing every step on the way. This is what `pwd` does: Above it said that I was in `/home/me`, telling me my exact position in the filesystem in an absolute manner; therefore paths starting with the root directory are called *absolute paths*.

³Here 'files' denotes data, directories, and links as it is the most generic concept.

⁴A short version of this can be found by typing `man 7 hier`.

Unfortunately, absolute paths can be very inconvenient and long. Remember that we said `ls` can be provided with a path to list the contents of any directory in the filesystem, not just the one you are currently in and that in the above example the current directory contained `Documents`, `Downloads`, `Pictures`, and `Private`. Imagine you have a subdirectory in `Documents` which is called `university`. To list it, you could type

```
$ ls /home/me/Documents/university
```

An awful lot to type I'd say! To remedy this situation you can provide a path relative to your current working directory, i.e. a *relative path*. `pwd` told us that our current working directory is `/home/me`, so the relative path to `university` is just `Documents/university` without a leading `'/'`. Every path starting with a slash will be interpreted as an absolute path, while relative paths just start with the name of the next directory you hop into from your current directory.

That's it for filesystem theory, now we'll enable you to help yourself on a (GNU/Linux) system.

3.3 Help Utilities

Every (GNU/Linux) system comes with a lot of built-in documentation installed by default and of course the Internet hosts millions of HOW-TOs, tutorials, etc. We're going to focus on the offline documentation, but also provide some hints where you usually get reliable information online.

This section can be treated as a reference – reading it now in its entirety may cause you a bit of a headache as you may not be familiar enough with commands, options and arguments right now. For the beginning it will suffice if you skim through this section to get a basic idea about which facilities exist.

3.3.1 `man`

`man`, which is a shorthand for “manual page” is the traditional UNIX documentation and knowing how to use it will help you not only on (GNU/Linux), but also on any other UNIX you'll encounter like the various BSDs. In case you ask someone who is in a hurry about a program and the only response is “RTFM”, meaning “Read The Fucking⁵ Manual”, this program is usually what they want you to use.

If you want to access the manual page of the `ls` program, all you have to do is to type:

```
$ man ls
```

After this the man page for `ls` should appear. To scroll down press `SPACE` and to exit press `Q`.

We won't go into a lot of details here because your first task now is to type

⁵Or “Fine”, for the very pure.

```
$ man man
```

This will fire up the man page of `man`. Read it to learn how to use this program efficiently and make sure to read the “EXAMPLES” section!

The main things you should remember from this is how to move within the man pages, how to exit them, what sections are and how you can tell the program to only return entries from certain sections. Useful options you may want to remember are

`-k` | `--apropos` List the programs whose name or short description match the keyword you searched for. Note that this is the same thing the `apropos` program does.⁶

`-a` | `--all` Show all man pages from all sections which match the keyword you searched for.

If you read through `man`’s man page you’ll understand what these options do ;)

3.3.2 `info`

While `man` is available on any UNIX system, the `info` pages were invented by the GNU project and are therefore not necessarily available. However, all widely used (GNU/Linux) distros have them installed by default – at least in parts. Unfortunately, the Debian project considers the license of the `info` files as nonfree and must thus be installed by hand using:

```
$ sudo apt install texinfo-doc-nonfree
```

Don’t ask...

To get the `info` page of `ls` type

```
$ info ls
```

You can again use `SPACE` to scroll forward and `Q` to get out of `info`.

Your next task is now to read

```
$ info info
```

As `info` is a bit more sophisticated than `man`, offering links, footnotes, menus, and other references it is essential to read this. Furthermore, `info` has a lot more keyboard shortcuts you can use, all of which are documented in its `info` page.

The difference between man pages and `info` pages is not a clearcut one. As a rule of thumb, man pages are often more concise and only useful if you already know what a program does in general, but just want to check available options, i.e. they are often very technical and offer few explanations.

The `info` pages, on the other hand, are often much more verbose, have sections for different kinds of options (while the man pages just list them one after the other) and at least try to be a bit more accessible to complete beginners.

⁶You guessed it: Use `man apropos` to find out more.

Just compare some man pages and info pages for the same program, e.g. `man ls` vs. `info ls` to see the differences and get a feeling for them.

Note also, that all documentation is written by humans and some are more capable of explaining how stuff works than others. For this reason the quality of man pages and info pages can vary greatly from program to program. A good example can be found in [this xkcd comic](#).

3.3.3 `--help` and `-h`

Sometimes you are just not sure about what was the right option to get what you want or you only forgot the correct spelling of the option. In these cases you may not want to read through the whole man or info page and most programs make this possible by providing a `--help` or `-h` option. So, typing `COMMAND --help` or `COMMAND -h` will in most cases print a short description of the command and its most common options to the screen.

Just try it out to see for yourself.

3.3.4 `help` and `type`

In [3.1](#) we mentioned that not every command you type is a full program, but some are just shell builtins. These are keywords understood by your shell. You can compare this with your GUI: Most icons you click on will start a new program, but some things are done by the GUI itself, e.g. if you click on the Windows button, it will display a list of programs you can start, but this list is just part of the GUI.

For these shell builtins no man pages or info pages exist. A command you'll use *very* often is `cd` ("change directory") which enables you to change your current working directory. But if you try to type

```
$ man cd
```

you'll get a man page titled "`BASH_BUILTINS(1)`" which will most likely be very confusing. To get help specifically for `cd` (or any other shell builtin for that matter) just type

```
$ help cd
```

and the help will be displayed on the terminal.

But, you may ask yourself, how am I supposed to know what is a program and what is a shell builtin?! Don't despair! This is where the `type` shell builtin comes into play. If you tried to access the man and/or info pages of a command and didn't get a satisfying result, try

```
$ type COMMAND_NAME
```

e.g.

```
$ type cd
```

```
cd is a shell builtin
```

And voilà, now you can be sure.

3.3.5 which and whereis

Many programs are continuously developed and change from one release to another; functionality may be added, reduced, removed, deprecated, etc. And at some points you may not know whether you want the old or the new version. One way to solve this is to simply install both, use them and then decide which one you like more. On a GUI this may be indicated by two different icons for the different versions, but how can you tell the difference on the command-line?

This is where the `which` and `whereis` utilities come in handy.

`which` tells you where in the filesystem the executable you are going to call is located and if you have installed two versions each of them will reside in a different place. Thus, if you type

```
$ which pwd
/bin/pwd
```

`which` tells you that if you call the `pwd` program, it will use the one in the `/bin` directory. So, in case you have a second version of `pwd` installed, which resides in `/usr/local/bin`, you'll have to use the absolute path to call it.

But sometimes you won't know if there are different versions of the same program on your machine, e.g. when you start working in a new place with your machine already set up for you. To find out your possibilities try

```
$ whereis COMMAND
```

which will show you where versions of `COMMAND` are installed, where their man pages are located and if the source code of the command is stored in a standard location on the system.

Attention: The following explanation will most likely make no sense at all if you haven't read about variables in 4.7. The actual difference between `which` and `whereis` is that the former looks through your `PATH` variable and returns the first executable it finds which actually is the one your shell would execute while the latter scans through your whole `PATH` as well as `MANPATH` plus some standard locations commonly used for programs on any (GNU/Linux) system, and returns everything it finds there, separated into executables, source files, and man pages.

3.3.6 whatis

The `whatis` command tells you what a command does on the most basic level, e.g.

```
$ whatis ls
ls (1)                - list directory contents
```

As you know by now, each man page has a one-line description at its start. It is exactly this one line which `whatis` will print.

3.3.7 The Internet Is Your Friend

If you have to do something real quick you may find yourself in a situation where you don't want to deal with man pages or info pages, but would prefer a ready-made command with all options you want already set. As it is very likely that other persons have had the problem at hand before you and many people post everything they achieve online, chances are that you can find the command of your dreams posted in some online forum, ready to be copy-pasted into your terminal emulator.

As you might expect, there are a myriad different sources where you can find information, but we will only deal with very few of them which offer reliable information in most cases.

Distros' Websites

Each distro has its website and often Wikis attached to them. Many have a lot of useful information, but keep in mind that these may be distro-specific. It is also important to know how your distro is organized; in the case of Debian, you are faced with a community project where the whole content is provided by people spending their spare time on providing information to others, but no one is responsible for doing so. This often leads to very helpful replies to questions, targeted at total beginners, but as no one is responsible for updating this content, sometimes you find very outdated information. Red Hat Enterprise Linux, on the other hand, is a distro aimed at corporations, providing a *lot* of information. However, as their material is intended for IT-professionals, you won't find a lot of explanation in many cases as they assume you already know what you are doing/want to do and just show you how you can do it with the tools Red Hat provides.

Thus, do a bit of research about each distro before using their documentation and how-tos.

Debian: Information about Debian can be found at:

- <https://www.debian.org>
- <https://wiki.debian.org/>
- <https://debian-handbook.info/>. This is the Debian Administrator's Handbook, the e-book version of which can be downloaded freely. However, you can also pay for the e-book version, and in case you start relying on it, it is definitely a good idea to give the authors something back for their hard work. The current (2019-08-22) version of the book is for Debian 8 Jessie, but the current Debian release is Debian 10 Buster, so some information may be outdated.

Ubuntu: Ubuntu is an enterprise distribution with a desktop and a server version. They have a very active community as well which answers a lot of questions in the forums.

- <https://www.ubuntu.com>

- <https://tutorials.ubuntu.com/>. You can filter tutorials for different categories, e.g. server or desktop.
- <https://help.ubuntu.com/>
- <https://ubuntuforums.org/>
- <https://askubuntu.com/> This is a searchable question and answer site.

Red Hat Enterprise Linux: As mentioned above, RHEL is an enterprise distribution and one of the most prominent examples showing how much money you can earn with free software. They have excellent, though rather technical guides for many topics, and some things can be only accessed if you have a support contract with Red Hat.

- <https://www.redhat.com/> Their main page is very much like the website of other companies – a lot of bragging and ads, but it’s still a good starting point to get a feeling about what Red Hat as a company is doing.
- <https://access.redhat.com/> This is where you start out if you look for help. Browse a bit; they have a lot of documentation, guides, etc. and many things can be downloaded as PDF for free.

CentOS: CentOS, the “Community Enterprise OS” is the community version of RHEL. It is only rarely suited for home Desktop use and the software it ships is very outdated in many cases (as all the good stuff is reserved for “actual” RHEL). However, if you want to get to know RHEL without paying for a license, CentOS is a good start and may be useful if you want to run small servers at home/in the cloud.

- <https://centos.org/>
- <https://wiki.centos.org/>. While this Wiki is still alive, it is updated rather sparsely and the current release CentOS 7 is not documented in any reasonable fashion. So much so, that <https://wiki.centos.org/docs/> just tells you to visit the above mentioned <https://access.redhat.com/>.

Arch Linux: While Arch Linux is considered to be rather complicated and suited only for experienced users, it has a thriving community and a very well maintained Wiki with in-depth explanations for a lot of things. Highly recommended if you want to know something in detail.

- <https://wiki.archlinux.org/>

We’ll stop here, but you can comb through all the other distros’ websites as well, which is a very useful thing to do before you change distro.

If you are interested how many distros are out there and which ones are the most heavily used, you may find <https://distrowatch.com/> (lists also non-Linux UNIX systems like the various BSDs) or <https://static.lwn.net/Distributions/> useful.

Other Online Sources

However, there are also sites dedicated to Q&A, i.e. you can post a question and someone in the community replies with a hopefully correct answer. The ones you'll encounter over and over again are:

Stack Overflow: To be found at <https://stackoverflow.com/>. If you did some programming already or are going to do so, this site will also pop up many times. Whenever you research some problem you have with the command-line it is almost certain that you are going to land on Stack Overflow after some time, regardless which search engine you use.

Stack Exchange for UNIX/Linux: Strictly speaking this is just a subsite of Stack Overflow and many topics specific to (GNU/)Linux and/or UNIX in general will can be found here: <https://unix.stackexchange.com/>.

With these tools at your disposal you should be able to solve almost any problem your new shiny (GNU/)Linux system throws at you. **BUT**, beware! Not everything people post there is correct and if you are unlucky, someone may even put some malicious code in their reply, so **always** think before you copy-paste something onto your command-line and press ENTER, especially if you have no idea what the command you want to use can do. Be even more careful if the line you copy starts with `sudo`. The good thing about UNIX systems is that you can do anything you like, but the bad thing is that you can do anything you type.

4 The Command-Line

Now, after so much theory, let's do something practical! The first thing to do is to start a terminal emulator if you haven't already done so.

Before we are going to start working with commands, we'll talk a bit about keyboard-shortcuts with which you can move quickly on a line. For this purpose just type a sentence on the command-line without pressing ENTER:

```
$ this is a beautiful test sentence
```

Your cursor will now be at the end of the line. If you made a typo somewhere in the line you can move there using the arrow keys, but this gets annoying pretty quickly as soon as the line gets long (your mouse won't work here). Thus, play around with the following shortcuts:¹

CTRL-A	Move to the beginning of the line.
CTRL-E	Move to the end of the line.
CTRL-K	Delete everything from your cursor to the end of the line.
CTRL-U	Delete everything from your cursor to the beginning of the line.
CTRL-T	Switch the sign currently below the cursor with the one before it and move the cursor one character to the right.
ALT-F	Move one word forward.
ALT-B	Move one word backwards.
ALT-T	Switch the word below the cursor with the one preceding it.

There are even more shortcuts, so look them up on your own if you're curious, but the ones above will suffice for the beginning. In case you don't see the use in some of them: That's ok, use cases will come ;)

And there is one more *vital* thing to know about moving around on the command-line: tab-completion. If you want to pass an argument to a program and you've already typed enough that the argument is uniquely identified, you can just hit TAB and the bash will automatically complete the argument. Why this is extremely useful will become clear once you start using commands and arguments, just keep this in mind. Additionally, if you press TAB twice, the bash will list all currently possible arguments. Try it out when following the instruction below!

¹Some of them may not work, because the terminal-emulator you use interprets these shortcuts differently (e.g. by opening up some menus). Usually this can be changed in the settings of the terminal-emulator.

4.1 Moving and Looking Around

Now, onwards to the command-line! For starters, let's find out where we currently are – you already know how to do this:

```
$ pwd
/home/me
```

Now, let's see what's in the directory we're currently in, which we can do with `ls(1)`, which “lists” directory entries:

```
$ ls
bin      documents  nikola     playground      unixcourse
Desktop  Downloads  personal   tor-browser_en-US
```

As you already know, you can pass options to `ls`. We'll start with the `-a` | `--all` option. This one shows you all files in the current directory:

```
$ ls -a
.          Downloads      .pki
..         .dvdcss        playground
.android   .F5Networks    .python_history
.ardentyst .gconf         .ssh
.bash_history .gnupg        tor-browser_en-US
.bash_logout .gnuplot_history .vim
.bash_profile .gphoto       .viminfo
.bashrc     .ICEauthority .vimrc
bin         .lessht       .wget-hsts
.cache      .local        .Xauthority
.config     .mozilla      .xsession
Desktop    .neomutt      .xsession-errors
.dmrc      .xsession-errors.old
```

What?! Where did all these files come from?! Don't worry, they have been here all the time, but they are *hidden files*. You see that those files which have not been listed before, start with a dot `'.'` and this indicate that they are ‘hidden’. This does not mean that you are not allowed to see them, but they are mostly configuration files which you just don't want to see every time you use `ls` as they just add a lot of useless output. For example, your browser will store your bookmarks, preferences, cookies, browsing history, etc. in a hidden directory. This way it does not confuse your profile with the one of another user who uses the same computer. Thus, a hidden file is just a file whose name starts with a dot, and programs like `ls` don't display them by default – nothing more to it. You can also create hidden files yourself if you start their name with a dot!

However, two hidden files are found in every directory: `'.'` and `'..'`. `'.'` is a pointer to the directory itself² and `'..'` is a pointer to the directory above this directory. This

²Why this is useful will become clear later.

is a very convenient shortcut if you want to change to the directory above the one you are currently in, but you don't want to type the absolute path from the root directory. Changing directories? Great, we're already at the next command: `cd` which very surprisingly stands for 'change directory'. It takes as an argument a path (relative or absolute) and changes your current working directory:

```
$ cd Documents
$ pwd
/home/me/Documents
$ cd ..
$ pwd
/home/me
```

You can see here that we changed into the `Documents` directory and from there one directory "upwards" by using `..` as shortcut. Otherwise we would have had to type `cd /home/me` to change from `/home/me/Documents` to the `/home/me`.

Two more things regarding `cd`:

- As we've seen in [3.3.4](#), it is a shell-builtin, so if you want to know how it works, you have to type "`help cd`".
- If you just type `cd` on the command-line and press `ENTER` without providing an argument, `cd` will change to your home directory. The home directory is where all your personal stuff is stored and in the examples above, my home directory is `/home/me`. Thus, if you get lost in the filesystem or you are in a hurry and just want to go back to your home directory, type `cd`, press `ENTER` and you'll be right there.

Now that you've seen `cd`, let's go back to `ls`. Combining `ls`, `pwd`, and `cd` the filesystem is yours to explore and we'll let you do that in a minute, just one more option to `ls` and the associated concepts:

With the `-l` option `ls` provides you with "long listing":

```
$ ls -l
total 36
drwxr-xr-x  2 me me 4096 Aug 27 14:53 bin
drwxr-xr-x  2 me me 4096 Sep  4 17:49 Desktop
drwxr-xr-x 14 me me 4096 Sep  7 17:40 documents
drwxr-xr-x  2 me me 4096 Sep 10 21:43 Downloads
drwxr-xr-x  8 me me 4096 Aug 24 20:05 nikola
drwxr-xr-x  4 me me 4096 Aug 27 14:54 personal
drwxr-xr-x  3 me me 4096 Aug 26 20:43 playground
drwx----- 3 me me 4096 Sep  9 14:36 tor-browser_en-US
drwxr-xr-x  2 me me 4096 Sep 17 11:48 unixcourse
```

As you can see there is a lot more information about each file. Actually, the output is now separated into fields, separated by a space. These fields are from left to right:

Permissions: See below, that's the new concept we're going to explain ;)

Hard Links: The number of hard links which point to this file. If you are interested in links, read it up, we won't cover them here as you don't need them for now.

Owner: The user to whom this file belongs.

Group: The group to which this file belongs.

Size: The size of the file in bytes.

Modification Time: The month, day and time give you the exact time when the file was last modified. If it was not modified in the current year, the year of the last modification is given as well.

Name: The name of the file.

Each file has an user owner which is usually the user who created the file in the first place as well as a group owner, which is usually one of the groups the user owner belongs to.³ But what does that mean?

Well, it has to do with the first field which consists of a rather bewildering number of letters and dashes at first sight. However, it is actually a very simple system and tells you what you can do with a file:

The first character tells you *what kind of file* this is, e.g. 'd' indicates a directory and '-' indicates a regular file. There are others as well, but these are the most common and important ones – if you want to know more look at the info page of `ls(1)`.

The next nine characters show the permissions of the file, i.e. who is allowed to do what with a file. These nine characters are actually 3 groups of 3 permission sets: The first three are the permissions of the owner⁴, the second three are the permissions for the group, and the third permission set tells you what anybody else (often called "world" or "other") is allowed to do.

The reading of permissions is pretty easy: If there is a letter, the permission is present, if there is a dash it is absent. The positions in each permission group from left to right tell you about:

r "Read permission": If this permission is set, it is allowed to read the file.

w "Write permission": If this permission is set, it is allowed to write something to the file, i.e. change it.

x "Execute permission": If this permission is set, it is allowed to execute this file – this is typically set for all programs, like `ls(1)`.

With this information at hand, can you decode this output:

³From now on we will just speak of "owner" to mean "user owner" and of "group" to mean "group owner".

⁴Yes, even owners are not necessarily allowed to do everything with the file!

```
-rwxr-x--x 1 nina hackers 2048 Jan 14 14:21 amazing-script
```

Try it, write down your solution and afterwards read the following explanation.

This is a regular file, which can be executed by the owner, the group, and everyone else, can be read by the owner and the group and written only by the owner. It belongs to user nina and to the group hackers. It is 2048 bytes in size (i.e. 2 KB), was last modified on January 14th at 14:21, and is called “amazing-script”.

As you’ve seen in 3.2, you can also pass absolute and relative paths as arguments to `ls` to tell it what to list. There is one thing you should be aware in this case: If the argument is a regular file, `ls` will list just the file, but if the argument is a directory, `ls` will show you the content of the directory:

```
$ cd unixcourse
$ ls file1
file1
$ ls contentdir
file_00  file_05  file_10  file_15  subdir04  subdir09
file_01  file_06  file_11  subdir00  subdir05  subdir10
file_02  file_07  file_12  subdir01  subdir06
file_03  file_08  file_13  subdir02  subdir07
file_04  file_09  file_14  subdir03  subdir08
```

Great, now you are ready for your tour through the filesystem! Use `ls`, `cd`, and `pwd` to look around – you are on a free software OS, there are no secrets (except if you don’t have permission to look at something, of course ;)). Check the overview of the filesystem in 3.2 again to find potentially interesting places to look at.

4.2 Working with Files

Up to this point we’ve just looked at files, examined their properties and moved around in the filesystem tree. This is all nice and fine, but maybe we want to change files’ properties, or add stuff to or remove something from the filesystem. This is what this section is about. And for now, `cd`⁵ into the `unixcourse` directory you’ve downloaded at the beginning of 3 to be able to follow all the examples below.

One of the things you may have already wondered about is: What if you want to have a directory for all of your lovely cat GIFs – how can you create such a directory? Well, the answer to this is `mkdir(1)` “make directory”. In its simplest form it just takes a single argument: The name of the directory you want to create. Note, that this can be a relative or absolute path:

```
$ ls
contentdir      grepfile      rmdirectory
dir1            hasfileinit   rminteractive
```

⁵Yes, you can use command’s names as verbs.

```

dir2             hugefile      sedfile
dir3             join1         sedfile.bak
'dir with spaces' join2         sortfile
file1            linefile      tablefile
file3            numericssort
filetodelete     onelinefile
$ mkdir newdir
$ ls
contentdir      grepfile      onelinefile
dir1            hasfileinit  rmdirectory
dir2            hugefile      rminteractive
dir3            join1         sedfile
'dir with spaces' join2         sedfile.bak
file1            linefile      sortfile
file3            newdir        tablefile
filetodelete     numericssort
$ cd newdir
$ pwd
/home/me/unixcourse/newdir

```

One thing about names: Whether you create directories as we do now, or other files which we'll do later, there are certain rules you should adhere to, to make your life easier: A file name (which includes directory names!) should consist of only letters of the English alphabet, underscores '_', dashes '-', and digits, and should always start with an underscore or a letter. Don't use special characters like ä, ß, ж, €, etc. and don't use spaces, tabs, newlines, etc.! This is just conventional, so you *can* use whatever you want, but it may make your life very inconvenient. In case you encounter filenames containing spaces, you will have to quote the name or escape the spaces using '\'. If you use `ls(1)` in the course directory you'll see that there is a directory called "dir with spaces". To change into it, you have three options:

```

$ pwd
/home/me/unixcourse
$ cd "dir with spaces"
$ pwd
/home/me/unixcourse/dir with spaces
$ cd ..
$ pwd
/home/me/unixcourse
$ cd 'dir with spaces'
$ pwd
/home/me/unixcourse/dir with spaces
$ cd ..
$ pwd
/home/me/unixcourse

```



```
$ cd dir\ with\ spaces
$ pwd
/home/me/unixcourse/dir with spaces
```

So you see, you can use either single or double quotes⁶, or just escape each space with a prepended backslash. Again, we recommend that you don't do this because it may break some programs and scripts. Don't get us wrong – we don't think that it's great that you have to stick with these conventions, but as most people do, it will make it harder for you to use others' solutions to problems you encounter.

After this little detour about naming, let's return to creating directories. Per default `mkdir` assumes that the whole path of the directory you want to create exists, except for the last part of the path, i.e. the part behind the last `'/'`. If you try to create a whole directory hierarchy, this will fail:

```
$ ls dir1
$ mkdir dir1/subdir/subsubdir
mkdir: cannot create directory 'dir1/subdir/subsubdir': No such file or directory
```

The first command creates no output because there is nothing in `dir1` and when we try to create a subdirectory `subdir` with a sub-subdirectory `subsubdir` we get an error message (not a very helpful one in this case, but we know what the problem is anyway). But surely, there must be a way to create such hierarchies, right? Of course! You just have to pass the `-p` | `--parents` option to `mkdir`:

```
$ mkdir -p dir1/subdir/subsubdir
$ ls dir1
subdir
$ ls dir1/subdir
subsubdir
```

Voilà, this time the whole path was created.

Nothing persists forever, so from creation we move directly to destruction: Whenever you want to delete a directory, you can use `rmdir(1)` “remove directory”. Like `mkdir` it takes one (or more) directories as arguments and deletes them. Ok, let's go!

```
$ rmdir dir1/subdir
rmdir: failed to remove 'dir1/subdir/': Directory not empty
```

Hm, that didn't work out as expected. But this is a good thing in this case. `rmdir` only removes directories if they are empty, i.e. there are no files (except `'.'` and `'..'`) in it. Why is this such a good thing? Well, now we're getting at something you should *really* remember: On the command-line *there is no undo!* If you delete something, it is gone for good and only people with a lot of training and expertise are able to recover files

⁶They are not the same in other contexts which we'll encounter later.

or directories you deleted.⁷ Therefore, `rmdir` prevents you from mistakenly removing a lot of your files with a single line.

However, if you have a situation like ours, where we have `dir1` which contains only `subdir` which only contains `subsubdir` which contains nothing and you want to delete `subdir` and `subsubdir` `rmdir` offers the same option as `mkdir` does, to delete such a chain of empty directories, i.e. `-p` | `--parents`:

```
$ rmdir -p dir1/subdir/subsubdir
```

This deletes all three directories in the given path. Now, recreate the `dir1` directory and move on.

As you might have guessed, `rmdir` removes only directories – just try it out with the following file, conveniently placed in your course directory:

```
$ rmdir filetodelete
rmdir: failed to remove 'filetodelete': Not a directory
```

Now that wasn't surprising at all. But how can we remove files? The answer to this is `rm(1)` "remove". Thus:

```
$ ls
contentdir      grepfile        onelinefile
dir1            hasfileinit     rmdirectory
dir2            hugefile        rminteractive
dir3            join1           sedfile
'dir with spaces' join2           sedfile.bak
file1           linefile        sortfile
file3           newdir          tablefile
filetodelete    numericssort
$ rm filetodelete
contentdir      hasfileinit     rmdirectory
dir1            hugefile        rminteractive
dir2            join1           sedfile
dir3            join2           sedfile.bak
'dir with spaces' linefile        sortfile
file1           newdir          tablefile
file3           numericssort
grepfile        onelinefile
```

This time, the file is gone. And remember what we said earlier about removing stuff? It's gone. Totally. Absolutely. No rebirth (without a lot of money). If you'd try to remove a directory with `rm`, this won't work out of the box:

```
$ rm dir1
rm: cannot remove 'dir1': Is a directory
```

⁷Their training is usually directly reflected in the rather huge amount of money you have to pay them to get your data back.

No surprises here. However, `rm` has some options we're going to talk about:

- `-r | -R | --recursive` With this option, `rm` removes files and directories recursively, i.e. if you pass a directory as an argument with this option `rm` will remove this directory and *everything* within it. Again, no undo. *NO UNDO!* Sorry, just sayin'. For the sake of an example:

```
$ ls rmdirectory
file_01  file_03  file_05  file_07  file_09
file_02  file_04  file_06  file_08  file_10
$ rm -r rmdirectory
$ ls rmdirectory
ls: cannot access 'rmdirectory/': No such file or directory
```

Yes, the directory and all ten files it contained are gone now.

- `-i` If you use this option, `rm` will ask you to confirm the deletion of each file, which can save your one file you forgot was in that directory, but *really* need.⁸ Thus:

```
$ ls rminteractive
file_01  file_03  file_05  file_07  file_09
file_02  file_04  file_06  file_08  file_10
$ rm -ri rminteractive
rm: descend into directory 'rminteractive/'? y
rm: remove regular file 'rminteractive/file_01'? y
rm: remove regular file 'rminteractive/file_05'? y
rm: remove regular file 'rminteractive/file_04'? n
rm: remove regular file 'rminteractive/file_09'? y
rm: remove regular file 'rminteractive/file_07'? y
rm: remove regular file 'rminteractive/file_10'? y
rm: remove regular file 'rminteractive/file_02'? n
rm: remove regular file 'rminteractive/file_03'? y
rm: remove regular file 'rminteractive/file_06'? n
rm: remove regular file 'rminteractive/file_08'? y
rm: remove directory 'rminteractive/'? n
$ ls rminteractive
file_02  file_04  file_06
```

For each question `rm` asks you have to type 'y' for "yes" or 'n' for "no".

- `-f | --force` Force `rm` to do what you told it remove everything without questions.⁹ While `-r | -R | --recursive` was dangerous, *this* one is where the real bad stuff

⁸Ever realized that "file" is just an anagram of "life"?

⁹In some cases `rm` asks you if you really want to delete a file even without the `-i` option present. See `man 1 rm`.

begins. Don't use it if you are not absolutely, 100%, utterly, totally, completely convinced that you know what you are doing. "Oh, come on", you may think, "I'm not *that* stupid, I know this by now. What can possibly go wrong?". Well, have a look at [this](#), where someone accidentally wiped all data of a whole company (including the backup) with one single `rm -rf`. The author of these lines once removed all invoices of a company he worked for, but fortunately there was no `-r` option present, so it were just the invoices of the current year. There was no backup. I had to retype them by hand from printed versions – great fun, 10 out of 10, would recommend...

So, what else can you do with files? What about copying stuff from one place to somewhere else? This can be done with the `cp(1)` command. Its basic syntax is:

```
cp [OPTION]... SOURCE DESTINATION
```

i.e. after maybe passing one or more options to `cd` you first tell it *what* you want to copy and then you tell it *where* to copy it. Source and destination can be relative or absolute paths. As a very simple example, you can do:

```
$ ls
contentdir      grepfile        numericssort
dir1            hasfileinit     onelinefile
dir2            hugefile        rminteractive
dir3            join1           sedfile
'dir with spaces' join2           sedfile.bak
file1           linefile        sortfile
file3           newdir          tablefile

$ cp file1 file2
$ ls
contentdir      grepfile        onelinefile
dir1            hasfileinit     rminteractive
dir2            hugefile        sedfile
dir3            join1           sedfile.bak
'dir with spaces' join2           sortfile
file1           linefile        tablefile
file2           newdir
file3           numericssort
```

However, there are some things about this simple scheme you have to take into account:

- If you want to copy a directory and all its content you have to use the `-r` | `-R` | `--recursive` option, like with `rm` when you wanted to delete a directory with its contents.
- If the destination is a directory, `cp` will copy the source into the destination directory, e.g.:

```
$ ls dir1
$ cp file1 dir1
$ ls dir1
file1
```

- If the destination is a directory, you can give multiple sources, i.e. you can copy multiple files at once into a directory by listing them all on the command-line and providing the directory you want to copy them into as the last argument:

```
$ cp file1 file2 file3 dir1
$ ls dir1
file1 file2 file3
```

- But, wait, we already copied `file1` to `dir1`, so which version is now in `dir1`? As you may have guessed from the behaviour of `rm` and `rmdir`: The command-line is unforgiving. If there is already a file with the same name in a directory, this file is overwritten and `cp` won't tell you about it – it assumes that you know what you are doing. Fortunately, just like `rm` there is a `-i` | `--interactive` option which asks you every time a file would be overwritten. In addition to that, `cp` also offers the `-n` | `--no-clobber` option which tells the program to never overwrite existing files.

The last but one thing we're going to cover for working with files is `ow` to rename and move them around. This is done by the same utility, `mv(1)` “move”. Its standard syntax is:

```
mv [OPTION]... SOURCE DESTINATION
```

Not only is its syntax the same as the one for `cp`, the same caveats about how to move multiple files and moving directories apply as well, with one exception: There is no `-r` | `-R` | `--recursive` or a similar option for `mv` – it always moves a whole directory if you pass one as source. If you just want to rename a file without moving it into another place in the filesystem, you just put the destination in the same directory:

```
$ ls
contentdir      file3           numericssort
dir1             grepfile       onelinefile
dir2            hasfileinit    rminteractive
dir3            hugefile       sedfile
'dir with spaces' join1          sedfile.bak
file1           join2          sortfile
file2           linefile       tablefile
$ mv file3 renamedfile
$ ls
contentdir      grepfile       onelinefile
```

dir1	hasfileinit	renamedfile
dir2	hugefile	rminteractive
dir3	join1	sedfile
'dir with spaces'	join2	sedfile.bak
file1	linefile	sortfile
file2	numericssort	tablefile

We'll end this section with something a bit more abstract again. In 4.1 we introduced you to the concept of permissions when you encountered the output of `ls -l`. Permissions were originally used in UNIX when many users sitting in front of terminals (the hardware devices you saw in 3.1) connected to central computers where all of them worked at the same time. To enable them to collaborate more easily they could set the permissions for group and others as they wanted to. This was done by the `chmod(1)` “change (file) mode”. `chmod`'s basic syntax is:

```
chmod [OPTION]... MODE[,MODE]... FILE...
```

There are two ways to pass a set of permissions to `chmod`: By a symbolic representation and by an octal number. The general format of the symbolic mode is:

```
[ugoa...][[-+=][perms...]...]
```

The first part indicates for whom you want to change the permissions: `u` for “user”, i.e. the user who owns the file, `g` for “group”, `o` for “others” and `a` for “all”. The default is `a`, i.e. to change the permissions for everyone, but if you write `u` you change only the user's permissions and if you write `ug` you change the permissions of the user and the group. As you may have noticed `ugo` and `a` are equivalent.

The second part means what you want to do: `+` to add the following permissions, `-` to remove the following permissions and `=` to set the permissions to exactly the following permissions.

The third field consists of the permissions you want to add/remove/set, i.e. the letters `r`, `w`, and `x`.¹⁰

Consider this example:

```
$ ls -l file1
-rw-r--r-- 1 me me 29 Sep 18 15:24 file1
$ chmod g+w file1
$ ls -l file1
-rw-rw-r-- 1 me me 29 Sep 18 15:24 file1
```

As you can see, we gave the group (`g`) additional (`+`) write permission (`w`), but left the other permissions intact.

Now it's your turn: Try to remove all rights from the group and the others. Is there more than one way to do this?

¹⁰If you look at the manpage of `chmod` you'll see that there are three more possible values, but for now, we'll ignore them.

We're not going to explain the octal mode here: It's not hard, but it can be confusing at the beginning and can take a bit more time to grasp – read the manpage if you are interested :)

While `chmod` changes the permissions on a file, there are also two utilities you can use to change the user owner and group owner of a file: `chown(1)` “change owner” to change the user owner and `chgrp(1)` “change group” to change the group owner. Their syntax is rather similar:

```
chown [OPTION]... [OWNER] [:[GROUP]] FILE...
chgrp [OPTION]... GROUP FILE...
```

As you can see, `chown` can actually be used to change user and group. In many cases where you use one of them you actually want to change all ownerships within a whole subtree of the filesystem. Thus, both of them offer a `-R` | `--recursive` to do so recursively. Note that this is an uppercase ‘R’!

This can be useful e.g. if someone copies a file from their home directory into yours, but the owner information is copied as well, so you have to change it before you can use it.

4.3 Working with Content

At this point you know how to move around and how to change files, but you haven't learned how to work with the stuff that is *in* files, i.e. content. Many files on your system contain human readable¹¹ text which you can easily manipulate.

4.3.1 Looking at Content

But before we start writing stuff somewhere, we'll look at it first. The easiest way to see the contents of a file is to use the `cat(1)` command. This is an abbreviation for “concatenate”, i.e. combining stuff. Its basic syntax is:

```
cat [OPTION]... [FILE]...
```

Now let's unravel the mysteries of the ominous `file1` with which we played around in the last section without ever knowing what's in there:

```
$ cat file1
A file with copiable content
```

Well, that wasn't very mysterious... It just contains a single line of text saying “A file with copiable content”. So that is what `cat` does: It reads the contents from a file and prints them to the screen. What about another file:

```
$ cat onlinefile
This is a nice one liner
```

¹¹Well, let's say “readable by *some* humans” ;)

Ok, also nothing very interesting. But what about this concatenation thing? Have a look:

```
$ cat file1 onelinefile
A file with copiable content
This is a nice one liner
$ cat onelinefile file1
This is a nice one liner
A file with copiable content
```

You see, `cat` concatenates the content of the files in the order you pass them on the command-line. This is tremendously useful if you want to e.g. combine multiple files into one very long file without copy-pasting all the stuff. How to do this will be shown later.

As you may have realized, if files have a lot of content, i.e. many pages of text, `cat` will be rather inconvenient because it just prints everything to the screen and you can't read fast enough to see everything:

```
$ cat hugefile
```

To read through such a file, you can use programs called “pagers” – they display text files one page at a time. The standard pager on (GNU/Linux) is `less(1)`.¹² So, let's view our file:

```
$ less hugefile
```

To move around scroll forward one page, use `SPACE`, for backwards use `B`. You can also move up and down one line with the arrow keys. To search for a pattern forward press `/` enter the pattern and press `ENTER`, to search for a pattern backwards, press `?` and press `ENTER`. When you finally want to quit the program, press `Q`.

But then again, how are you supposed to know beforehand how big a file is, i.e. whether it makes sense to use `cat` or `less`? There are different strategies and one you know already:

```
$ ls -l onelinefile hugefile
-rw-r--r-- 1 me me 431888 Sep 18 17:52 hugefile
-rw-r--r-- 1 me me    25 Sep 17 13:53 onelinefile
```

If you look at the size column you can see that there is a huge difference between these two files. Unfortunately, using bytes as a measure is not very human-readable, especially because even `hugefile` could in fact consist of only a single line – a *very* long line, but it is possible. Therefore, it would be useful to be able to view the lines of a file. This can be done with `wc(1)` “word count”:

¹²In the old days, the standard pager was `more(1)`. When someone wanted to replace it they thought that “less it more” and thus the name of `less`.


```
$ wc onelinefile
 1  6 25 onelinefile
$ wc hugefile
7654 70552 431888 hugefile
```

`wc` provides with some useful *metadata*, i.e. data about data, about a file. The first column is the number of lines in the file, the second column shows the number of words in the file and the third column tells you how many bytes the file contains. You can also list just one of these units using the `-l` | `--lines` for only the line count, `-w` | `--words` for only the word count, and `-c` | `--bytes`¹³ for only the byte count.

4.3.2 Creating Content and I/O redirection

After so much information reception it is time to let you create something yourself! And the first tool for this is `echo(1)`. Very unexpectedly `echo` echoes what you pass to it:

```
$ echo Hello command-line
Hello command-line
```

This seems easy enough. But don't be fooled, `echo` is an extremely useful tool and will introduce you to a lot of new concepts. Just as an aside: It is often a good idea to put the stuff you want `echo` to echo in double quotes. Read the man page to find out why this is so ;)

The first one is the concept of *I/O streams*¹⁴. A stream is a flow of data made available through time. While this sounds rather abstract, you've already worked with streams all the time in this course! Whenever you open a `bash` there are three standard streams opened for this `bash` as well:

`stdin` or "Standard Input", i.e. the stream into which you put data. This is usually your keyboard because this is where you input the stuff you write. The `bash` then reads those characters from the keyboard and displays them on the command-line.

`stdout` or "Standard Output", i.e. the stream into which the data goes. In most cases this is your screen or rather the terminal emulator on your screen. Yes, most of the time a program produces output, it writes it to `stdout`!

`stderr` or "Standard Error", i.e. the stream into which error messages are sent. This is also mostly your screen and you've already seen this in action: Remember when we tried to remove a non-empty directory with `rmdir`? There was an error message displayed and while you saw it on your screen, this was actually written to `stderr` and not to `stdout`.

¹³"Why `-c`?" you may ask. This is because years ago characters were always encoded in ASCII (see the [Wikipedia entry](#) or, even better [ascii\(7\)](#) for an explanation) where each of them fit into one byte and so the `-c` for character came into being.

¹⁴For Input/Output stream.

But why make two stream which write to the same thing?! The reason is simple: They do so usually, but you can manipulate them individually. Manipulate? Oh yeah, you can play around with those streams until your head spins! This called I/O redirection.

It is time, to create your first file! You know now that whenever echo echoes something back to you, it actually reads input from stdin and puts it to stdout. But what if we *redirect* stdout to a file? See here:

```
$ ls
contentdir      grepfile        onelinefile
dir1            hasfileinit     renamedfile
dir2            hugefile        rminteractive
dir3            join1           sedfile
'dir with spaces' join2           sedfile.bak
file1           linefile        sortfile
file2           numericssort    tablefile
$ echo "I want this in a file" > echofile
$ ls
contentdir      grepfile        renamedfile
dir1            hasfileinit     rminteractive
dir2            hugefile        sedfile
dir3            join1           sedfile.bak
'dir with spaces' join2           sortfile
echofile        linefile        tablefile
file1           numericssort
file2           onelinefile
$ cat echofile
I want this in a file
```

So now you know that by using ‘>’ you can redirect stdout to a file. Let’s try that again:

```
$ echo "More content" > echofile
$ cat echofile
More content
```

Ok, that was not expected. Where’s the former line gone? Well, it’s not here anymore. If you use ‘>’ to redirect something to a file, the file will be *truncated*, i.e. totally emptied, and afterwards the new content will be written to the file. If you want to just add something to a file, without deleting what is already in there, you have to use “>>”, which *appends* data to a file:

```
$ echo "And now a new line" >> echofile
$ cat echofile
More content
And now a new line
```

Note that there is no way to put something in the middle of a file using just the command-line. To do this you need a text editor which we’ll discuss later.

You can also use this redirection on any other command, e.g. if you want to store its output so you can look at it later:

```
$ ls -l > lsfile
$ cat lsfile
total 504
drwxr-xr-x 13 me me 4096 Sep 19 2018 contentdir
drwxr-xr-x 2 me me 4096 Aug 24 03:44 dir1
drwxr-xr-x 2 me me 4096 Sep 17 2018 dir2
drwxr-xr-x 2 me me 4096 Sep 17 2018 dir3
drwxr-xr-x 2 me me 4096 Sep 17 2018 dir with spaces
-rw-r--r-- 1 me me 32 Aug 24 09:18 echofile
-rw----- 1 me me 29 Sep 18 2018 file1
-rw-r--r-- 1 me me 29 Aug 24 03:45 file2
-rw-r--r-- 1 me me 472 Sep 23 2018 grepfile
-rw-r--r-- 1 me me 0 Sep 23 2018 hasfileinit
-rw-r--r-- 1 me me 431888 Sep 18 2018 hugefile
-rw-r--r-- 1 me me 46 Sep 23 2018 join1
-rw-r--r-- 1 me me 48 Sep 23 2018 join2
-rw-r--r-- 1 me me 405 Sep 23 2018 linefile
-rw-r--r-- 1 me me 0 Aug 24 09:21 lsfile
-rw-r--r-- 1 me me 26 Sep 23 2018 numericssort
-rw-r--r-- 1 me me 25 Sep 17 2018 onelinefile
-rw-r--r-- 1 me me 11 Sep 23 2018 renamedfile
drwxr-xr-x 2 me me 4096 Sep 18 2018 rminteractive
-rw-r--r-- 1 me me 331 Sep 23 2018 sedfile
-rw-r--r-- 1 me me 331 Sep 23 2018 sedfile.bak
-rw-r--r-- 1 me me 18 Sep 23 2018 sortfile
-rw-r--r-- 1 me me 747 Sep 23 2018 tablefile
```

Your `ls` output has now been stored in a file.

Now, what if we want to do something with `stderr`?

```
$ ls nonexistentfile
ls: cannot access 'nonexistentfile': No such file or directory
$ ls nonexistentfile > errorfile
ls: cannot access 'nonexistentfile': No such file or directory
$ cat errorfile
$ ls nonexistentfile 2> errorfile
$ cat errorfile
ls: cannot access 'nonexistentfile': No such file or directory
```

You can see that if we use `>` to redirect the output, `errorfile` stays empty, because `ls` didn't print anything to `stdout`, just to `stderr` and because we just redirected `stdout` to the file, `stderr` is still printed to the command-line. The redirection for `stderr` is

“2>” and you can see that using this results in the error message being not printed to the screen, but written to the file instead.

To be honest, we cheated a bit: Actually the streams are numbered, starting with zero: `stdin` is 0, `stdout` is 1 and `stderr` is 2. You can only write to `stdout` and `stderr` because they are output streams while `stdin` is an input stream you can’t write to.¹⁵ Thus, only output streams can be redirected using ‘>’ and you can prepend the number of the output stream you want to redirect, i.e. “1>” or “2>”. If you don’t specify anything, however, “1>” is implied because you much more often want to redirect `stdout` than `stderr` and this is why just ‘>’ above worked as it did. And before we forget: These numbers work the same way when you want to append stuff, i.e. “1>>” is the same as “>>”, and “2>>” appends `stderr` to the given file. This is useful e.g. if you want a program to log all error messages in a file.

But we talked about “manipulating streams individually”. What does that mean?

```
$ ls file1 nowhere
ls: cannot access 'nowhere': No such file or directory
file1
$ ls file1 nowhere > lsfile 2> errorfile
$ cat lsfile
file1
$ cat errorfile
ls: cannot access 'nowhere': No such file or directory
```

As you can see, it is possible to redirect `stdout` and `stderr` to different files. This can be very useful if you have a program with a lot of output and you want to keep normal output and error messages apart.

Note that if you want both of them to redirect to the same file you have to first redirect one stream to the file and then redirect the other one to the other stream:

```
$ ls file1 nowhere > twostreams 2>&1
$ cat twostreams
ls: cannot access 'nowhere': No such file or directory
file1
```

To redirect `stderr` to `stdout` you use `2>&1` which reads approximately as “redirect `stderr` to where `stdout` writes to”. You could also do the following:

```
$ ls file1 nowhere 2> twostreams 1>&2
$ cat twostreams
ls: cannot access 'nowhere': No such file or directory
file1
```

The result is the same. Note, however, that the order is important: *First*, redirect one stream to a file, and *afterwards* redirect the other stream to it. This makes sense: If you

¹⁵This is pretty easy to understand if you think about the meaning of the streams: How would you try to write to your keyboard?

read the sentence describing `2>&1` carefully, you'll notice that it says "where stdout writes to" and not "to stdout". Thus, if you redirect stderr to stdout first, and afterwards redirect stdout, stderr writes to where stdout wrote to before you redirected it.

To finish this, we'll give you a little shortcut: If you don't want to care about this first and second thing and consider "`> filename 2>&1`" an awful lot to type, you can use the short version: "`&> filename`". The above line then becomes:

```
$ ls file1 nothere &> twostreams
```

This is much more convenient, but introduces a new caveat: Not all shells are the same and this shortcut is something that works in bash but does not necessarily do so in other shells you may encounter. Just keep this in mind if you run into trouble ;)

In 4.3.1 when we introduced you to cat we mentioned that you can use it to combine contents of multiple files into a single one. Now you have all the necessary tools; play around with cat and I/O-redirection :)

4.4 Archiving and Compression

Sometimes it is useful to put files into a collection called an *archive*. An example would be if you want to send multiple files via email, but you don't want to attach 10 files. Put them all in an archive and you only need to append a single file.

Another thing that is useful in many cases is *compression*. Compressing data means to encode it in a way that uses less bits than the original data. This is possible by using algorithms which make use of redundant data – most data has such redundant elements because it comes in certain file formats which demand structure, i.e. patterns.

4.4.1 Archiving with tar

To create an archive of multiple files on (GNU/)Linux and many other UNIX-like systems you can use the `tar(1)` "tape archive" utility.¹⁶ You can tell by its name that this program is pretty old, as it was created to work with tapes.

`tar` has an incredible amount of functionality (have a look at the manpage) but we will cover only the most basic stuff here. If you need some more advanced archiving stuff, do some research – the chances are pretty good, that `tar` is capable of fulfilling your needs.

To create an archive you can use:

```
$ tar -cvf myfirstarchive.tar file1 file2 hugefile
file1
file2
hugefile
$ file myfirstarchive.tar
myfirstarchive.tar: POSIX tar archive (GNU)
```

¹⁶If you come from the world of Microsloth Windoze you may already be familiar with ZIP archives.

We used `file(1)` here, which is a program that basically tells you what kind of file something is. The explanation of the options is quite straightforward:

`-c` | `--create` tells `tar` to create an archive.

`-v` | `--verbose` tells `tar` to be verbose, i.e. to print each file it processes to `stdout`.

`-f` | `--file=ARCHIVE` tells `tar` how it should call the archive. It is conventional to add the `.tar` extension to `tar` archives, though not strictly necessary. However, do yourself a favour and use this extension ;)

After you told `tar` what you want and where to put it¹⁷ you have to put all files you want to be in the archive on the command-line. If you pass a directory to `tar` it will automatically put all files in the directory in the archive as well, so you don't have to tell it explicitly to do that like you had to with `cp` or `rm`.

To check that everything worked correctly, you can check which files are in an archive by using the `-t` | `--list` option:

```
$ tar -tf myfirstarchive.tar
file1
file2
hugefile
```

This works essentially like `ls` but for archives.

However, if you received an archive, the relevant thing you want to do is not just to list the files but to *extract* them on your system. To try this, create a directory, change into it and extract the files there using the `-x` | `--extract` | `--get` option:

```
$ mkdir tar-dir
$ cd tar-dir
$ ls
$ tar -xvf ../myfirstarchive.tar
file1
file2
hugefile
$ ls
file1 file2 hugefile
```

It is always a good idea to list the contents of an archive you downloaded from the Internet before extracting it, to make sure that it really contains the files you want and not something else. Particularly, *never* extract an archive which contains files with absolute paths! This is something only malware and very careless people do, so you very likely don't want anything from such entities on your machine.

¹⁷I.e. the name of the archive. This can be an absolute or a relative path.

4.4.2 Compression Utilities

While archiving today is done mostly with `tar`, there are several utilities to compress files. Each of them uses a different algorithm to compress the data, but their usage is very similar. The only thing you should know about the algorithms used is how efficient they are in order to decide which one will be most appropriate for the problem you are facing.

One important fact you should know before starting to use compression is the following: Don't compress data which is already in a compressed format! If you do this you'll end up with a file that is *bigger* than the original one. Why is this? Well, as pointed out above, compression utilizes redundancies in data to store the same information in less bits. It also adds a bit of metadata to make recovering of the original data possible. If you have compressed data once and your compression algorithm is decent, there will be no redundant bits left to remove. Thus, if you now apply compression again, the algorithm will find nothing it can compress, but still adds its metadata, meaning you have more data than before.

For this reason, it is a good idea to check if the format of the files you want to compress already uses compression – well known examples are `.pdf`-files, `.odt`-files, and even `.docx`-files. You can try this out for yourself after you've learned how to use the utilities.

`gzip(1)`

The first tool we're going to look at is `gzip(1)`. Its basic syntax is:

```
$ gzip FILENAME...
```

By default `gzip` will do the following to each file you pass to it:

- Compress the data and store it in a file with the same name as the original plus the `.gz` extension.
- Remove the original file.

This especially the second point is important to keep in mind – if you want to keep your old file(s), use the `-k` | `--keep` flag.

To decompress a compressed file you have two options:

1. Use the `-d` | `--decompress` | `--uncompress` option.
2. Use the `gunzip` command which is more or less just a shortcut of the above.

By default this restores the uncompressed file and deletes the compressed one afterwards – again you can use `-k` | `--keep` flag to change this behaviour.

However, sometimes you just want to see what's in a compressed file without storing the decompressed version. To do this, you have two options again:

1. Use the `-c | --stdout | --to-stdout` flag which little surprisingly just prints the contents of the file to stdout.
2. Use the `zcat` utility which is again just a shortcut for the above.

Demo time!¹⁸

```
$ ls -lh hugefile
-rw-r--r-- 1 me me 422K Sep 18 2018 hugefile
$ gzip hugefile
$ ls hugefile
ls: cannot access 'hugefile': No such file or directory
$ ls -lh hugefile.gz
-rw-r--r-- 1 me me 163K Sep 18 2018 hugefile.gz
$ gunzip hugefile.gz
$ ls hugefile.gz
ls: cannot access 'hugefile.gz': No such file or directory
$ ls -lh hugefile
-rw-r--r-- 1 me me 422K Sep 18 2018 hugefile
```

You can see that the original file was compressed to less than half of its size! However, you can do even better: Compressing data takes time and time is often precious. The better the compression the longer it takes, which can be a problem. Therefore `gzip` provides an option to adjust the speed vs. compression quality. You can pass it a number between 1 and 9 preceeded by a dash to indicate you want the program to be really fast (-1) or compress really good (-9). The default level is -6. Have a look:

```
$ ls -l hugefile
-rw-r--r-- 1 me me 422K Sep 18 2018 hugefile
$ gzip -1 hugefile
$ ls -l hugefile.gz
-rw-r--r-- 1 me me 192K Sep 18 2018 hugefile.gz
$ gunzip hugefile.gz
$ gzip -9 hugefile
$ ls -l hugefile.gz
-rw-r--r-- 1 me me 162 Sep 18 2018 hugefile.gz
```

So the best compression produces a file which is about 15% smaller than what the worst compression creates. This may seem minor to you, but think about data centers which have to store hundreds of thousands of terra bytes of data where this can make an enormous difference.

And now a quick demonstration of `zcat`:

¹⁸The `-h | --human-readable` option to `ls(1)` makes the size output more readable by adjusting the units; e.g. 1024 bytes will be shown as 1K instead to indicate that this is one KB.


```
$ cat file1
A file with copiable content
$ gzip file1
$ ls file1.gz
file1.gz
$ zcat file1.gz
A file with copiable content
$ ls file1.gz
file1.gz
$ gunzip file1.gz
```

You can see that in this case the compressed file stays intact and just its content is output to `stdout`. The last step restores `file1` as it will be used in exercises later in the course.

Of the compression utilities we're going to discuss here, `gzip` is the “worst” one in terms of compression, but it is still widely used and “worst” in this case means still very good – and fast!

`bzip2(1)`

The `bzip2(1)` command has pretty much the same syntax and default behaviour as `gzip`, and there are a `bunzip2` as well as a `bzcat` utilities. Compressed files get the `.bz2` by default.

However, not all things are totally identical, so read the man page of `bzip2` and play around a bit to compare the compression rates of `bzip2` and `gzip`. In general, `bzip2` offers better rates than `gzip` but worse than the following utility.

`xz(1)`

Again, syntax and behaviour are very similar to the above and there are `unxz` and `xzcat` utilities offering the same functionality as with the other two programs. By default compressed files get the `.xz` extension.

This is the best and most versatile compression tool of the ones we discuss which will become apparent if you read its much longer man page and see what options it offers.

Again, play around to get a feeling for the program.

4.4.3 Combining Archiving and Compressing Data

You've seen that you can put multiple files into an archive which is a single file and you know that you can compress files – wouldn't it be convenient to combine these features, i.e. to create an archive with all the files you want and then compress the whole archive in one step? The good news is: You can. The better news is: `tar` has options to combine this into one single step.

-z | --gzip What this option does depends on the arguments you pass to tar. If you have given the -c option, tar will first create an archive and then compress the whole thing in one step:

```
$ tar -cvzf files.tgz file1 file2 renamedfile
file1
file2
renamedfile
$ ls files.tgz
files.tgz
$ tar -xvzf files.tgz
file1
file2
renamedfile
```

Conventionally gzip compressed tar-archives use .tar.gz or less verbose .tgz as extension.

-j | --bzip2 This compresses/decompresses a tar archive using the bzip2 utility. Such archives usually get the .tbz or .tbz2 extension. Again you can also use .tar.bz2 if you want to be more verbose.

-J | --xz To your surprise, this will create a xz-compressed archive. The conventional extension is .txz in this case.

Note that if you use compression with tar the original files stay intact; just like when you use tar to create an archive.

4.5 Textfiles, Pipes, Wildcards, and a Little Bit of Regex

Here we enter the last big section about working with files and their content on the command-line, so take a deep breath and get ready to learn some incredibly useful concepts and tools!

All of them are mainly concerned with working with textfiles. You've already encountered echo, cat, less, and wc, but we'll add many more and find new applications for some of them as well.

4.5.1 Of Heads and Tails

Sometimes you want to look only at the beginning or the end of a file, e.g. if you do some C programming you may just want to see the #includes and comments at the top of a source file, or you've written a $\text{\LaTeX}$ file and forgot which packages you included, but you want them in a new project, so again, you only want the first few lines. To do this, there is the head(1) utility, whose basic syntax is:

```
$ head [OPTION]... [FILE]...
```

It prints the “head” of a file to `stdout`, which by default means the first 10 lines of each file you passed to it as an argument. However, 10 is sometimes not enough (or too much), so you can easily control the number of lines being printed with the `-n | lines=NUM` option:

```
$ head linefile
Line 01
Line 02
Line 03
Line 04
Line 05
Line 06
Line 07
Line 08
Line 09
Line 10
$ head -n 5 linefile
Line 01
Line 02
Line 03
Line 04
Line 05
$ head -5 linefile
Line 01
Line 02
Line 03
Line 04
Line 05
```

As you can see, you can leave out the `-n` and write the desired number directly after the dash.

On the other hand, you may have added something to a file and don’t remember what it was, or if you are running a web server or anything else that creates logs it is often useful to just look at the last lines of a textfile. The utility of choice in this case is `tail(1)`. Its syntax is equivalent to the one of `head` and by default it prints the last 10 lines of a file to `stdout`.

There is one common use case for `tail`, however, which is not used with `head`: We mentioned programs creating log files, i.e. files which log their activity. These files increase through time as the programs append more and more data to the end of the logs. If you want/have to monitor such programs, it is often useful to watch the logs while growing, but of course you only want to see the new entries at the end of the file. This can be done using `tail` with the `-f | --follow` option. To test this, do the following:

1. Type `tail -f linefile`.
2. Open a second terminal window and change to the `unixcourse` directory.
3. Append a new line to `linefile`:

```
$ echo "newline" >> linefile
```

4. In the window which runs `tail -f linefile` you will see that a new line has been added.

Now try what happens, if you type ‘>’ instead of “>>”.

To stop `tail` from running, go to the terminal window it runs in and press `CTRL-C`. This will terminate the process. We’ll explain this to you in [4.6](#), don’t worry.

4.5.2 Sorting Text and Using Pipes

You’ll encounter many situations where you want to sort the content of a file – e.g. a list of words you want to sort alphabetically or some numbers representing measurements and you want to quickly check their range. This is what the very fittingly named `sort(1)` program is good for:

```
$ cat sortfile
T
B
A
W
c
N
B
I
P
$ sort sortfile
A
B
B
c
I
N
P
T
W
```

That was easy enough! And now for numbers:

```
$ cat numeric-sort
9
253
85
76
0010
1000
$ sort numeric-sort
0010
1000
123
253
76
85
9
```

What?! Most likely this did not correspond to your expectations. How can `sort` think that 1000 is less than 9?? Of course, `sort` is not that stupid. The problem is caused by different expectations: `sort` sorts text alphabetically and numbers are just part of the alphabet: They come before all letters and there are exactly 10 of them: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Thus, 1000 and 9 are just words – 1000 starts with 1 which precedes 9 and must thus be placed first, like `abcd` precedes `z`. That is what alphabetically sorted means: you order everything according to their first letter, and only then you look at the second one to order those with the same first letter further, etc.

“OK”, you may say, “this may be an answer, but it is still pretty annoying.” Correct. And because this problem is very common, `sort` offers the `-n` | `--numeric-sort` option which tells it to sort the lines of a file numerically:

```
$ sort -n numeric-sort
9
0010
76
85
123
253
1000
```

Much better!

In the above output, which sorted the lines of `sortfile` alphabetically the letter B appeared twice. This is all nice and fine in many situation, but sometimes you want only the unique entries in a file, e.g. if you want to check how many different measurements you had or how many different users accessed your web server. If you do some research, you’ll find that this is what the `uniq(1)` program does. If you read its man page you’ll find the following:

Note: ‘`uniq`’ does not detect repeated lines unless they are adjacent.

What this essentially means is, that you have to be sure that all duplicate lines in a file are adjunct before passing the file to `uniq`. One possibility to achieve this is to sort the file first. Using what you’ve just learned and I/O redirection, you can do the following:

```
$ sort sortfile > tmp
$ uniq tmp
A
B
c
I
N
P
T
W
```

Great, there is only one line now starting with B. However, this workflow is rather clumsy. And worse: Imagine the file whose unique entries you want has 10 or even 100 GB – this approach forces you to create another file of the same size just for this little task.

Surely, there must be more intelligent way to do this! There is: Pipes.¹⁹

A pipe is a unidirectional data channel which allows you to “pipe” the output of one program as input to another program, i.e. `stdout` of one program will be sent to `stdin` of another program.²⁰

To pipe the output of one program to another one, you use the vertical bar ‘|’:

```
$ sort sortfile | uniq
A
B
c
I
N
P
T
W
```

It does not look like much, but this is one of the most powerfull abilities the command-line offers because you can combine arbitrarily many commands in this manner, each one changing the output and passing it to the next. Try to find out what the following line does and what output you expect before running it:

```
$ sort sortfile | uniq | head -n 5 | tail -n 3
```

Or what about this:

¹⁹For those of you eagerly reading man pages who have already found the `-u` | `--unique` option for `sort`: Yes, this works perfectly fine, but for now, pretend you have no idea ;)

²⁰Actually, pipes can do more than just this, but for our purposes, this is totally sufficient.

```
$ head -n 200 hugefile | less
```

You can do an incredible lot of things with piping and we're going to encounter more use cases soon.

Note that you pipe `stdout` to `stdin`, leaving `stderr` alone. This means that if something in your pipe goes wrong, you'll see the error messages printed to the screen:

```
$ ls file1 file2 file3 file4 | head -2
ls: cannot access 'file3': No such file or directory
ls: cannot access 'file4': No such file or directory
file1
file2
```

4.5.3 Wild Cards and Gentle Regexes

Until now you passed each argument you wanted a command to process directly to it, only using tab completion to increase your speed.

But, what if you have a directory with hundreds of files and you don't even know all of their names, but you do know that all of them contain some string or have a common number in their name? In this case you may want to use *wildcards*. Wildcards are characters the shell treats as special and which enable you to look for patterns. They are:

- * The asterisk matches any string, including the null string, i.e. the empty string.

- ? The question mark matches any single character.

- [...] This pattern matches any one of the enclosed characters. There are several things to consider here:

- Special characters lose their special meaning within brackets.
- If the first character is '^' the meaning of the brackets change to "match any characters *not* in the given set", e.g. `[^ab]` matches anything except for 'a' and 'b'.
- You can use ranges within brackets, e.g. `[0-9]` matches any number between 0 and 9, `[a-z]` matches lower case characters, and `[A-Z]` upper case.
- If you want to match a special character in your bracket expression, there are some simple rules:
 - To match a '^', just put it somewhere else than the start, e.g. `[a^]` matches 'a' or '^'.
 - To match a '-' put it as last or first character, e.g. `[ab-]`.
 - To match a closing bracket, put it at the first position (or after the caret if you want to negate it).

But what does that mean? Let's look at some examples:

```

$ ls *
contentdir      file1           join2           rminteractive
dir1            file2          linefile       sedfile
dir2            files.tgz      lsfile        sedfile.bak
dir3            grepfile       myfirstarchive.tar sortfile
'dir with spaces' hasfileinit    numericssort   tablefile
echofile        hugefile.gz    onelinefile    tar-dir
errorfile       join1          renamedfile    twostreams

$ ls *file
echofile  grepfile  lsfile      renamedfile  sortfile
errorfile linefile  onelinefile sedfile      tablefile

$ ls file*
file1 file2 files.tgz

$ ls *file*
echofile  files.tgz  linefile  sedfile
errorfile grepfile  lsfile    sedfile.bak
file1     hasfileinit onelinefile sortfile
file2     hugefile.gz renamedfile tablefile

$ ls file?
file1 file2

$ ls [flsn]*
file1 files.tgz lsfile      sedfile      sortfile
file2 linefile  numericssort sedfile.bak

```

Try to explain the output yourself and only read on afterwards!

The first example just passes a ‘*’ to `ls` and as this matches any pattern, all files in the directory are listed. The second example matches all files whose name ends in `file`. The third example matches all those whose name begins with `file`. The fourth example matches any file whose name contains the string `file` somewhere. The fifth example matches any file whose name starts with `file` followed by exactly one additional character. The last example matches any file whose name starts with `f`, `l`, `s`, or `n`.

Wildcards can be very handy to match only certain files but not others. However, they are restricted to the command-line and still very coarse grained filters. What if you want to check if a pattern exists e.g. at the end of lines in a file? This can be achieved using a *regular expression*, or *regex* for short.

Regular Expressions and `grep(1)`

First, let’s shortly introduce you to the `grep(1)` utility. This is the program of choice if you want to “grab” some text patterns in a file. Using `less` and `cat` you know how to look at everything that’s in a file and in combination with `sort`, `uniq`, `head`, and `tail` you’ve learned to remove some information, but until now you can’t look for some specific patterns in a file, e.g. is there a measurement of exactly “3.14159265358979”

or a sentence which contains the word “octothorpe” in the file?

With `grep` you’re able to do this. Its basic syntax is:

```
grep [OPTION]... PATTERN [FILE]...
```

and the program looks for `PATTERN` in each file you pass to it. So, let’s see if this measurement we talked about above is found:

```
$ cat grepfile
this line contains a hashtag "#"
a hash "#" can be found in this line
but only here you'll find a wild octothorpe "#"
A line ending in ZZZZ
aaaaa
aaaa
aaa
a
and for the curious: hash == hashtag == octothorpe

// this is single line C comment

This one ends in asdZ
# a comment in bash!
```

(parentheses)(but nothing in between).
The quick brown fox jumps over the lazy dog.

```
Starting spaces.
and this one in zzzz
on (this line) someone inserted (rather random) parentheses
$ grep '3.14159265358979' grepfile
$ grep 'octothorpe' grepfile
but only here you'll find a wild octothorpe "#"
and for the curious: hash == hashtag == octothorpe
```

The initial `cat` command is just shows you what’s in the file. The others tell you that there is no line containing the beginning of π in this file, but two contain the word “octothorpe”. If you also want to know which line(s) contain matching patterns, use the `-n` | `--line-number` option to `grep`. Note that you should always enclose the pattern you pass to `grep` in single quotes because otherwise, shell wild cards may get in your way. We’ll see an example of this later.

Very well, but what about this regex stuff? The pattern you pass to `grep` can be a regex. Using regexes is similar to using wildcards, but it is *much* more powerful. To get the official definition for POSIX²¹ regular expressions you can read `regex(7)`, but

²¹“Portable Operating System Interface X”. The ‘X’ was appended because it was intended for UNIX-like OSs.

we'll do it a bit less formal.

As with wildcards, certain characters in regexes do not have their literal meaning. These are called *metacharacters*. The metacharacters for basic regular expressions are:

Table 4.1:

<code>^</code>	The caret matches the beginning of a line.
<code>\$</code>	The dollar sign matches the end of a line.
<code>.</code>	The dot matches any single character (similar to the '?' wildcard).
<code>[]</code>	As with wildcards, a bracket-expression matches any single character within the brackets. Note that ranges in regexes work as you expect, i.e. <code>[a-z]</code> matches all lowercase letters. The caret, dash, and closing bracket can be included the same way as with wildcards.
<code>[^]</code>	A bracket-expression starting with a caret matches any single character <i>not</i> found within the brackets, <code>[^abc]</code> matches anything except a, b, and c.
<code>*</code>	The asterisk in a regular expression matches the preceding element 0 or more times. Note that this is different from the meaning of the asterisk as a wildcard! Using wildcards, the expression <code>b*</code> matches everything that starts with the letter 'b', but as a regular expression it matches 0 or more 'b's!

With these new knowledge, try the following using `grep` with `grepfile`:

- Print all lines which contain a '#'.
- Print all lines which begin with a '#'.
- Print all lines which end in a punctuation mark.
- Print all lines which end in 'Z' or 'z'.
- Print all empty lines.
- Print all lines which contain no 'a'.

Check your results by looking at `grepfile` with `cat` or `less`. If you can solve all of them: Great! If not: Don't despair, it takes time to get used to the concept of regexes.

There is also a very useful component to `grep` and regexes if you like to play crossword puzzles: On many distros, you can find at least one word list of the language your system runs on in `/usr/share/dict`. The most common one is just called "words". On my system it contains 102401 words. Suppose now that you are looking for a mountain

range consisting of 12 letters, the second one is a ‘p’, and there fifth, fourth, and third to last letters are “hia”:

```
$ grep '^p.....hia..$' /usr/share/dict/words
Appalachia's
Appalachians
```

As an aside: If you ever want to look for one of the metacharacters as a literal, e.g. you want to know which lines contain a ‘\$’ you have to escape them with a backslash, i.e. you have to type “\\$” in your pattern.

However, regexes don’t end here. There are some more metacharacters than those we’ve see so far. These are used for so called extended regular expressions and we’ll cover them now:

Table 4.2:

()	The expression between the parentheses is seen as a single element from the outside, e.g. (abc)* matches 0 or more occurrences of the string “abc”, e.g. it matches “abc”, “abcabc”.
\n	Matches the n^{th} expression delimited by parenthesis, e.g. (ab)(cd) . . \2\1 matches any string that starts with “abcd” followed by two characters followed by “cd” followed by “ab”.
{m,n}	Matches any string that contains the preceding expression at least m and at most n times, e.g. q{2,5} matches 2, 3, 4, or 5 consecutive ‘q’s. The version of grep that is installed on most (GNU/Linux) distros also allows some variations of this scheme: If you pass only one number n without a comma, the expression matches occurrences of exactly n times the same character, e.g. d{3} matches exactly “ddd”. You can also leave out one of the numbers but provide a comma to give just a lower or upper bound, e.g. i{,10} indicates “up to 10 consecutive ‘i’s” while i{10,} reads as “at least 10 consecutive ‘i’s”.
?	Matches the preceding expression zero or one times, e.g. friends? matches “friend” or “friends”. Like with the asterisk, this differs from the question mark if it is used as a wildcard! The wildcard ‘?’ is the same as the regex ‘.’.
+	The plus sign matches one or more occurrences of the preceding expression, e.g. w+ matches ‘w’, “ww”, “www”, etc. The difference to the asterisk is that the plus sign does <i>not</i> match zero occurrences.

Continued on next page

Table 4.2: (Continued)

	The vertical bar matches either the preceding or the following expression, e.g. <code>ab Bc</code> matches “abc” and “aBc”. Of course, this could also be done by <code>a[bB]c</code> , so why care about ‘ ’? Well, if you combine it with parentheses or braces, it becomes clear: <code>(sw) m (gr)eet</code> matches “sweet”, “meet”, or “greet” or <code>(Ada) (A\).</code> Lovelace matches “Ada Lovelace” and “A. Lovelace”. Or think about something like <code>a{2,3} e{1,5}</code> .
--	--

Unfortunately, with these metacharacters things are a bit confusing: While you had to use a backslash if you wanted the literal meaning of the metacharacters we talked about before, with this group it is the other way around: `grep` (and other tools) interpret them literally, *unless* you prepend a ‘\’. For example, ‘+’ is interpreted as a plus sign, you have to type “\+” to get the regex meaning. This can be pretty annoying, especially as this behaviour is not consistent: Some tools treat all metacharacters as metacharacters. And to make things really funny: If you pass the `-E` | `--extended-regexp` to `grep`, it will also interpret all metacharacters as metacharacters!

However, this will become less confusing as soon as you start using regexes regularly. Try to do the following, and play around with the `-E` switch:

- Print all lines which contain at least one ‘Z’ or ‘z’.
- Print all lines which contain expressions enclosed in parentheses. Then try to print all lines where there are two expressions in parentheses which follow directly after each other. And finally try to print only those lines where the parentheses-expression have some text between them.
- Print all lines containing the word “fox” or the word “bash”.
- Print all lines which consist exclusively of ‘a’s.
- Print all lines which consist exclusively of 3 or 4 ‘a’s.
- Print all lines which consist exclusively of at least 3 ‘a’s.

Check you results by looking at `grepfile` with `cat` or `less`.

After all metacharacters being covered and some practice we come to where we said, regexes and shell wild cards may get in their way. See the following example:

```
$ grep file? grepfile
grepfile:this line contains the string "file1"
$ grep -E 'file?' grepfile
this line contains the string "file1"
whereas here, we have file2
Another file present ist called "file23"
```

What happened here? In the first example you passed “file?” to grep without single quotes. Remember, that the shell calls any command you type and passes on the arguments you provide. However, in order to make wild cards possible, the shell first substitutes all wild cards expressions with their expansions and because “file?” matches the files file1 and file2 in the current directory, the shell actually calls

```
$ grep file1 file2 grepfile
```

which tells grep to look for the pattern “file1” in file2 and grepfile. This is the reason why grep prepends grepfile: in response to the first command because if you pass more than one file to search the program by default gives you the name of the matching file and the matching line. Sounds rather confusing? Well, then just use single quotes for patterns you pass to grep and such problems will never bother you :)

Now that you know some basic stuff about regexes and know the basics of grep, let's move on to two other programs which use regexes extensively:

sed(1) and awk(1)

These two utilities are much more powerful than those you encountered before, especially because awk is in fact a full-blown programming language, but we'll introduce only the very basics.

sed which is an acronym for “stream editor” is a tool with which you can edit a stream of textdata and print it to stdout (or redirect it to a file, of course). The usual way to invoke sed is:

```
sed [OPTION]... -e 'SCRIPT' [FILE]...
```

where SCRIPT is a command you pass to sed that tells it what to do. sed will then apply this command to each line of the input file. Sounds very confusing, but let's work through an example:

```
$ cat sedfile
```

```
This text talks about birds. Why? Because we all know that birds are
the best animals. All birds are much more beautiful and intelligent than
any other species (including humans). No one can ever doubt that birds
are superior. And one day, birds will take over the world! Thus, don't
ever mistreat a bird, it will get its revenge!
```

What utter nonsense! Birds?! Ridiculous! Surely it's moles who'll subdue the world in the near future! This text must be adjusted:

```
$ sed -e 's/bird\(s\)\?/mole\1/g' sedfile
```

```
This text talks about moles. Why? Because we all know that moles are
the best animals. All moles are much more beautiful and intelligent than
any other species (including humans). No one can ever doubt that moles
are superior. And one day, moles will take over the world! Thus, don't
ever mistreat a mole, it will get its revenge!
```

Much better! But how to dissect the command to `sed`?

Table 4.3:

<code>s</code>	This is the command which means “substitute” whose syntax is <code>s/REGEX/REPLACEMENT/FLAGS</code> There are other commands, like <code>d</code> for delete or <code>a</code> for append something.
<code>/</code>	The slash is used here as a delimiter character. This is just conventional and you can use any other character as a separator. Note, however, that in case you want your delimiter appear in the <code>REGEXP</code> or <code>REPLACEMENT</code> you have to escape it using a backslash.
<code>bird\s\)?</code>	The pattern to match. ²⁵

Table 4.4:

<code>mole\1</code>	The replacement. The <code>\1</code> refers to the first expression in parentheses in the pattern before, so if there is no ‘s’, it will be empty and if there is, it will print one.
<code>g</code>	This is a flag meaning “global”. If you don’t use it, <code>sed</code> only substitutes the first match of the regex on a line and leaves the rest alone. Just repeat the command above but delete the <code>g</code> to see what happens.

Phew, that’s a lot to congest. But once you get the hang of it, `sed` can be immensely useful.

A word of caution about redirection: Only in rare cases do you want to see the changes `sed` made just on the command-line; usually you want them to be stored somewhere and in many cases you want to replace the old file with the new content. Beware, the following will not work in the way you may expect:

```
$ sed -e 's/bird\s\)?/mole\1/g' sedfile > sedfile
```

If you do this, `sedfile` will be empty afterwards. But why? When you invoke `sed` and tell it to redirect its output to `sedfile` the first thing it does is to open this file for writing and because you used the greater-than operator it will truncate the file (i.e. empty it) and only then starts working on the now empty file. If you used the `>>` operator, the changed text would be appended to the current file, but the old content would still be there – most likely also not, what you wanted. This leaves you with two possibilities:

1. Redirect the output to a temporary file and use `mv` afterwards. This has the big advantage that you can check your changes before making them final.²⁶

²⁶It is in general a good idea to first output `sed`’s results to `stdout` or pipe them into `less` to check if the

2. Use sed's `-i [SUFFIX] | --in-place[=SUFFIX]` option. This tells sed to edit files in place. If you supply a SUFFIX, sed will create a backup of your original file with this suffix, which you can use to restore the old file if something has gone wrong.

sed can also be used to print only certain lines from a file, regardless of any patterns. This can be done using the `p` command and the `-n` option: If a line doesn't match anything you specified in a command, by default sed will print this line unchanged. To suppress this behaviour and show you only those parts of the file that sed works on is what the `-n | --quiet | --silent` option is for. The `p` command is the "print" command and can be prepended by a line number, e.g.:

```
$ sed -n -e '1p' sedfile
```

This text talks about birds. Why? Because we all know that birds are

prints only the first line of sedfile. We'll use this in [4.5.4](#).

After sed, we'll have a quick glance at awk. The awk program is actually an implementation of the AWK programming language which owes its name to its creators Alfred Aho, Peter Weinberger, and Brian Kernighan, but after learning it, some people have concluded it must come from "awkward". We won't go into any details here, just be aware that it exists and look at the following example:

```
$ awk '{ print "line" NR ": " $0; }' sedfile
```

```
line1: This text talks about birds. Why? Because we all know that birds are
line2: the best animals. All birds are much more beautiful and intelligent than
line3: any other species (including humans). No one can ever doubt that birds
line4: are superior. And one day, birds will take over the world! Thus, don't
line5: ever mistreat a bird, it will get its revenge!
```

You can see that this command prepended the word "line", the line number, stored in the variable (see [4.7](#)) NR and a space in front of each line. The current input line is stored in the variable \$0.

If you are totally confused now: That's OK, sed and awk are rather advanced tools, but as they can be immensely useful it is great to know about them and what they do in general, so you can come back and learn them in-depth if you ever need to. There are whole books written just about these two programs and other books that just treat regexes, so there is *a lot* to learn, definitely more than we can cover in this introductory workshop.

4.5.4 Cutting and Joining

Before we leave you alone with textfiles, there are two more things to talk about. The first has to do with cutting stuff up and joining it together.

You'll often have files which consist of columns and lines, representing some kind of table. Have a look at tablefile:

changes work as intended.

```
$ cat tablefile
12.9 2.6 9.9 5.9 9.5 7.9 1.2
13.6 9.3 7.8 1.9 6.7 5.7 8.2
13.1 7.4 1.0 3.7 3.0 1.8 13.1
8.4 12.3 13.1 12.8 10.1 7.3 11.9
1.5 9.3 10.4 2.6 9.6 1.1 12.8
5.8 11.6 7.1 5.2 1.7 1.8 9.6
8.7 2.9 11.8 2.7 3.7 5.4 10.6
2.7 2.7 3.6 13.0 4.2 10.8 1.8
9.6 5.9 5.7 9.5 9.5 3.1 11.2
10.9 1.3 3.0 4.9 8.2 7.7 6.6
13.0 12.0 3.6 13.5 7.8 12.4 4.9
1.6 12.5 1.4 6.9 10.2 3.7 3.8
7.3 5.1 11.2 12.5 1.0 10.3 3.1
6.6 12.2 5.4 2.2 13.9 12.5 12.2
8.7 2.4 10.9 6.7 5.9 5.2 11.8
11.4 7.2 1.3 12.2 4.7 3.2 10.7
3.4 7.2 7.4 12.8 4.9 6.4 9.2
3.9 12.7 3.2 3.3 6.7 8.0 8.5
9.5 12.8 6.8 2.7 7.2 2.4 13.7
12.4 3.6 2.1 3.1 10.9 7.7 12.8
10.4 1.2 5.4 2.6 7.0 7.5 8.8
8.9 7.3 3.6 6.3 7.2 3.8 7.4
12.9 12.7 8.2 11.1 2.5 5.3 10.3
11.2 6.4 11.9 7.3 2.6 10.5 3.4
```

This file consists of 24 lines and each line has 7 columns. This could e.g. be the output of a program which took measurements for 1 week, every hour, e.g. the second value in the first column represents the measurement at 1am (we started at 00:00) on Monday. All nice and fine, but the file is almost unreadable for a human. This is, where `cut(1)` comes in: Its purpose is to cut out columns while leaving out others and its basic syntax is:

```
cut OPTION... [FILE]...
```

It has some very useful options:

- `-d | --delimiter=DELIM` This tells cut by which character fields in the file are separated. In the case above they are separated by spaces, but it could be any other character as well.
- `-f | --fields=LIST` This tells cut which fields it should output.
- `--output-delimiter=STRING` If you want the output to be delimited differently than the input, you can use this option.

Thus, if you want just the the measurements of Wednesday, you can do:


```
$ cut -d" " -f3 tablefile
```

```
9.9
7.8
1.0
13.1
10.4
7.1
11.8
3.6
5.7
3.0
3.6
1.4
11.2
5.4
10.9
1.3
7.4
3.2
6.8
2.1
5.4
3.6
8.2
11.9
```

Or what about the files from Tuesday to Thursday:

```
$ cut -d" " -f2-4 --output-delimiter="^T|^T" tablefile
```

```
2.6      |      9.9      |      5.9
9.3      |      7.8      |      1.9
7.4      |      1.0      |      3.7
12.3     |      13.1     |      12.8
9.3      |      10.4     |      2.6
11.6     |      7.1      |      5.2
2.9      |      11.8     |      2.7
2.7      |      3.6      |      13.0
5.9      |      5.7      |      9.5
1.3      |      3.0      |      4.9
12.0     |      3.6      |      13.5
12.5     |      1.4      |      6.9
5.1      |      11.2     |      12.5
12.2     |      5.4      |      2.2
2.4      |      10.9     |      6.7
7.2      |      1.3      |      12.2
```

7.2		7.4		12.8
12.7		3.2		3.3
12.8		6.8		2.7
3.6		2.1		3.1
1.2		5.4		2.6
7.3		3.6		6.3
12.7		8.2		11.1
6.4		11.9		7.3

This looks much better!²⁷ If you want to see only the measurements taken between 10am and 1pm, you can combine this with `sed` and pipes:

```
$ sed -n -e '11,14p' tablefile | cut -d" " -f2-4 --output-delimiter="^T|^T"
12.0      |      3.6      |      13.5
12.5      |      1.4      |      6.9
5.1       |     11.2      |     12.5
12.2      |      5.4      |      2.2
```

First we select the hours we need and then we cut out the columns for Tuesday, Wednesday, and Thursday.

Another thing you may want to do is joining two files which share a common field. To achieve this, you can use `join(1)`:

```
$ cat join1
Price ArtNr
10€ 1
20€ 2
100€ 3
400€ 4
$ cat join2
ArtNr Name
1 pendrive
2 cable
3 screen
4 laptop
$ join -1 2 -2 1 join1 join2
ArtNr Price Name
1 10€ pendrive
2 20€ cable
3 100€ screen
4 400€ laptop
```

So, you tell `join` that the second field in the first file (`-1 2`) and the first field in the second file (`-2 1`) are identical and it should use them to join the files. There are some other options and if you're interested play around a bit :)

²⁷The `^T` in the command-line above means a tab-character. You can reproduce it on the command-line by typing `CTRL-V` followed by a tab.

4.5.5 Editors (a.k.a. Religions)

The last step on our journey through textfiles leads us into the fiercely disputed field of text editors. As the title of this section suggests, there are people who vehemently insist that the editor they use is the only true one and all the others should be forbidden. It may well be editors about which the first flamewars were fought and they are fought until today with absolutely no progress or use.²⁸

Our stance is: Taste differs and it's no good arguing about it, but we want you to be aware that you may encounter people who'll attack you just because your editor is not theirs.

One reason why editors may be such a emotional topic is that the really powerful ones take a considerable time to get used to and a lifetime to master. Thus, as soon as you learned one of those you may be really proud of you on the one side and feel like a total beginner if you encounter one of the others, which often leads to aversive reactions.

There are editor which need a GUI while others can work directly within a terminal. Common GUI editors on (GNU/Linux) systems are:

KWrite The KDE-Desktop's lightweight editor.

gedit The GNOME-Desktop's standard editor. If you left all the defaults intact when you installed the GUI of you VM at the beginning of the course, gedit will be installed on your system and just called "Text Editor".

There are gazillions of others – if you're interested look around and try some of them until you find something.

But as this course is about the command-line, we're emphasising on editors you can use in a terminal.

nano This very small and intuitive editor is now installed by default on many (GNU/Linux) systems.

vi(m) **vi** pronounced "Vee-Eye" and **vim** which stands for "Vi IMproved" and thus just pronounced "vim" are one of the main editors used by many programmers and sysadmins alike. They are incredibly powerful and have tons of features.

emacs The archrival to **vi(m)** for a long time, this giant of an editor is often described as "actually an operating system which happens to include a text editor". If you put in the effort to learn to use it efficiently, you'll be very productive when working with text.

Instead of talking about features of these editors, we'll let you learn them by their own means. First do the following:

```
$ sudo apt install vim emacs
```

²⁸You think we're joking? Have a look at [this](#).

This will install both editors. Now choose which one you want to try out first.

If it is vim, type:

```
$ vimtutor
```

And you will be in an interactive tutorial, showing you the basics of vim.

In case you chose emacs, type:

```
$ emacs
```

After the editor started, type CTRL-H followed by T. The tutorial will start.

We won't say anything else about editors; it is important to know how to use one of them, but which one doesn't really matter. Try a few, play around and stick with what you like most :)

4.6 Processes

After so much textfiles, we start with something completely different. You have worked a bit with commands by now, and maybe you wondered at some point, what they are. In very general terms we speak of a program as an executable binary that can be run on a CPU. Whenever you enter a command, the corresponding binary is loaded into memory and then each instruction is executed by the CPU. As soon as a program runs, it is called a *process*, i.e. a process is a program in execution. To make the difference a bit clearer: You have only one program found at `/bin/ls` but you can run it in different terminals at the same time, thus creating multiple processes from a single program.

If you want to see the processes currently running on your system the most widely used tools are `ps(1)` and `top(1)`. Their main difference is that `top` is interactive while `ps` is not. To view all processes that are currently running from the terminal you are working on, you can do:

```
$ ps
  PID TTY          TIME CMD
 2351 pts/4    00:00:00 bash
 4963 pts/4    00:00:00 ps
```

This tells you that there are just two processes associated with this terminal: your `bash` and the `ps` process which has already finished as soon as you can read the output. The displayed fields have the following meaning:

PID This is the “Process ID”. Each process is assigned a number that is unique on the currently running system, to enable the system (and you) to refer to it unambiguously – the name alone would not be sufficient.

TTY This tells you with which terminal²⁹ the process is associated. As you are working in a terminal emulator and not on a real terminal the “pts” in the output means “pseudo-terminal”.

²⁹TTY stands for teletypewriter, a rather old acronym.

TIME This tells you how many CPU time the process has used cumulatively.

CMD Rather obviously, this is the command's name.

To see all processes currently running on the system, use:

```
$ ps -elf
```

This creates a listing of all processes (-e) in a long (-l) and full (-f) format to show a lot of extra information.³⁰ We don't print it here, as the output is almost always more than 100 lines.

ps has a myriad of options and tweaks available, so it is highly recommended to read the man page. Additionally ps is a very old program and has acquired a lot of legacy-stuff during the time. Thus, there are many options which do almost-but-not-exactly the same, there are options with no dashes, one dash, or two dashes, etc. Therefore, reading the man page in this case also provides you with a glance of history.

top is also a very old program, but it is still very useful. If you start it the process will fill your screen and update periodically. The first line of the output will look something like this:

```
top - 14:11:33 up 4:48, 6 users, load average: 0.02, 0.02, 0.00
```

This tells you the current time, how long the system is running, how many users are currently logged in and the load average. Note that a load average of 1 means one CPU is fully loaded, i.e. if your computer has more than one CPU or more than one core, a load of more than 1 is no problem.

By default, processes are sorted according to the %CPU column, which tells you how much CPU they are using.

To quit the program type Q.

Again it is a good idea to read the man page to get acquainted with the many things you can do with top.

The processes you've seen so far run for only a very short time, e.g. the output of ls appears immediately and you can issue the next command. However, in 4.5.1 we already encountered a program which didn't return to command-line on its own, i.e. when we used tail -f. While this program runs until you tell it to stop, there are other programs which just take a long time and it may be very annoying if they block your command-line all the time. Consider starting the Firefox web browser from the command-line: It would be pretty bad if you had to choose between browsing and working.

Thus, there are a lot of tools to control processes. First of all, there are keyboard shortcuts which send signals³¹ to processes. Signals tell a process what to do, e.g. SIGTERM tells the process to terminate. The keyboard shortcuts you can use are:

³⁰You'll find the options aux (without a dash!) doing something very similar in a lot of forums as these are the traditional options.

³¹If you want to know more about which signals are available on (GNU/Linux) systems, you can read signal(7). And if you are into programming: They are a handy tool for inter-process communication.

Table 4.5:

CTRL-C	Send an interrupt signal (SIGINT) to the process currently blocking the command-line. Usually, this leads to the process being terminated, as you learned in 4.5.1 when you stopped the <code>tail -f</code> process.
CTRL-Z	Sends a stop signal (SIGTSTP) to a process. A stopped process won't react to anything, unless you send it a SIGCONT (for "continue") signal.

Some processes just need a long time or even don't have a defined point in time when they stop, e.g. compiling a large program takes a long time and your web browser should only stop when you tell it to do so. How can you use such programs without sacrificing your command-line? The easiest way is to start them with an ampersand appended on the command-line, e.g.

```
$ firefox &
[1] 5133
$
[1]+  Done                  firefox
$
```

As you can see, Firefox is started, the shell tells you that it has job number 1 and PID 5133. The ampersand tells the shell to start the process in the background and the next time you press ENTER it will tell you that the job has finished. However, if you could e.g. start a long backup job, put it in the background and the shell will inform you as soon as its done, even if this takes hours or days (provided you don't shut down your computer or close the shell during this time as that would kill the job).

The job number is also a unique number; not on the whole system, but just on the current shell. As it is usually much smaller than a PID it is also easier to remember for humans. To have a look how many jobs are currently associated with your bash, you can use the `jobs` shell-builtin.

4.6.1 Process Control

Now that you know how to find out which processes are running, either on the system or within your shell session and you know that you can start them in the background and are able to send two signals to them using the keyboard.

However, there is more you can do with processes. The bash offers some builtin functionality for this:

If you have a job running in the background, but you want it back on your shell, you can use the `fg` builtin, e.g.

```
$ xclock &
[1] 5257
```

```

$ jobs
[1]+  Running                  xclock &
$ fg 1
xclock
^Z
[1]+  Stopped                  xclock
$ bg 1
[1]+ xclock &
$ ps
  PID TTY          TIME CMD
  5130 pts/5        00:00:00 bash
  5257 pts/5        00:00:00 xclock
  5264 pts/5        00:00:00 ps
$ kill 5257
$
[1]+  Terminated              xclock

```

In this example you start an `xclock` in the background. Then you put it into the foreground using `fg 1` where 1 is the job ID of the process. After this, you lost access to your shell. The `^Z` in the output above indicates pressing CTRL-Z, i.e. sending a SIGTSTP to the process. After this you'll realise that `xclock` will no longer interact with you – the process is frozen. To let it continue in the background, you use the `bg` builtin. Afterwards you find out `xclock`'s PID and send it a signal using `kill`.³²

`kill` is one of the most important tools when doing process control. Read its man page to understand what it does and how. One last caveat: If you can avoid it, don't ever use `kill -9` to terminate a process. This will end the process immediately without giving it the possibility to clean up (close open files, remove temporary files, store your session, etc.), leaving it in a messy state which can lead to strange behaviour when you start it again. The only situation where killing a process in this way is when it doesn't react to anything else anymore and hogs a lot of resources you need.

4.7 Variables and Your Environment

We've covered a lot of ground already and by now you know a lot of commands and other stuff. However, one thing we didn't talk about yet is how does the shell know

³²This is a bit of a mess: There is a program called `kill(1)` which will be explained if you type "`man kill`". However, consider the following scenario: You have opened so many processes that you are not allowed to start new ones and some of them are no longer reacting, so you have to terminate them by sending them signals. However, `kill(1)` is a program and if you try to invoke it, this will be refused as you're not allowed to start any more processes. Therefore, there is a `bash`-builtin called `kill` which does exactly the same, but as it is just a part of `bash` it does not need to start a whole new process and will still be able to terminate the misbehaving ones. Thus, if you just type `kill` on the command-line the `bash`-builtin will be used and not the program `kill(1)`. This has the further advantage that you can use job IDs instead of PIDs. To tell `kill` that you're passing a job ID to it, prepend the number with `'%'`, i.e. in the above example: `kill %1`.

which program to execute if you pass a name like “ls” to it? The answer: There is a variable in your environment telling it what it needs to know.

4.7.1 Variables

If you’ve already done some programming it will most likely suffice if you skim through the following explanations to grasp how the shell handles variables.

A variable is just a name you give to a value. Consider the following Python snippet:

```
x = 6
y = 7
z = 6 * 7
print(z)
```

There are three variables in this snippet, `x`, `y` and `z`. The first two are directly set to a specific value while the last is set to the product of others and then printed to the screen. You know this stuff also from school, where you worked a lot with variables.

However, a variable doesn’t have to be a number in programming – it can be set to a string, a file descriptor, an instance of a class, etc.

In the context of the shell, variables are usually strings, e.g.:

```
$ myvar="Hello, World!"
$ echo $myvar
Hello, World!
$ x=6
$ y=7
$ z=$x*$y
$ echo -e "x == $x\ny == $y\nz == $z"
x == 6
y == 7
z == 6*7
```

There are several things to consider in this example:

- Variables in the shell are set by writing “name=value” which is little surprising. There *must not* be any spaces between name, =, and value!
- To reference the value of a name, prepend a ‘\$’ to the name.
- To print the content of a variable use the shell builtin `echo` or `echo(1)`.
- The shell interprets variables as strings and doesn’t perform mathematical operations by default.³³

³³There is a way to set `z` to the product of `x` and `y`, but it’s rarely used so we won’t discuss it here.

And about the `echo` statement: The `-e` option tells `echo` to interpret certain backslash-escaped characters, one of which is `'\n'`, the new line character.

One variable that has been with you this whole course is `PS1`. Not very recognizable, is it? Quite the opposite, this is the variable in which your prompt is stored, i.e. the `"user@machine $"` string that greets you on every line of input. Have a look at it:

```
$ echo $PS1
\[e]0;\u@\h [\W]\a\]${debian_chroot:+($debian_chroot)}\u@\h: [\W]\$
```

Ok, this seems rather complicated. But the basics are easy! The following elements in this string are useful to remember:

`\u`: Your username.

`\h`: The hostname, i.e. the name of your computer.

`\w`: Your current working directory as absolute path.

`\W`: Only the last element of your current working directory, e.g. if you are in `/home/me/unixcourse` this would be `unixcourse`.

You can do much, *much* more with your prompt, including setting colours, showing the time, giving you various information about the directory you are in, etc. For now, try something easy like:

```
$ OLDPSP1=$PS1 # backup your old prompt
$ PS1="> "
> PS1="\u > "
me > PS1="\u@\h \w $ "
me@machine ~/unixcouse $ PS1=$OLDPS1
$
```

First, we stored the original value of `PS1` in a new variable to make restoration easy.³⁴ Then we set the prompt to different values and restored it at the end.

As a starting point to what else you can do with your prompt, have a look at the [Arch Linux wiki entry](#).

There is a lot more to variables in the shell (and different shells offer different possibilities), but the above is sufficient for our purposes.

4.7.2 Your Environment

Variables in the shell are predominantly used to configure your environment or in shell scripts which will be covered later. But what is your environment?

It is the context in which you execute commands. Above we raised the question how the shell knows which programs to execute when you pass it a command and said there is a variable for this. This variable is named `"PATH"`. Let's have a look at it:

³⁴By the way, the shell will ignore everything you type after a `'#'`, so this can be used for comments.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

You can see that this is a list of absolute paths to directories, separated by colons. If you type a command, your shell looks in each of these locations whether there is a program of this name and executes the first match. Let's take `ls` as an example: First, we check where `ls` is located:

```
$ which ls
/bin/ls
```

Ok, it's in the `/bin` directory. Thus, if we type `ls` the shell does the following:

1. Look for "`ls`" in `/usr/local/bin`. Not found.
2. Look for "`ls`" in `/usr/bin/`. Not found.
3. Look for "`ls`" in `/bin`. Found!
4. Execute `/bin/ls`.

So far so good, but how can this be helpful to you? `PATH` is a variable and you've seen above that you can set variables yourself. Thus, you can add and remove directories in which you want your shell to look for executable commands. An easy example:

```
$ OLDPATH=$PATH
$ PATH=""
$ ls
bash: ls: No such file or directory
$ PATH=$OLDPATH
$ ls
contentdir      file1           join2           rminteractive
dir1            file2           linefile       sedfile
dir2            files.tgz       lsfile         sedfile.bak
dir3            grepfile        myfirstarchive.tar sortfile
'dir with spaces' hasfileinit     numericssort   tablefile
echofile        hugefile.gz     onelinefile    tar-dir
errorfile       join1           renamedfile    twostreams
```

Above we first stored the original value of `PATH` in another variable for easy recovery. Then we set `PATH` to an empty string and try to execute `ls`. Little surprisingly this doesn't work, because now the shell has nowhere to look for executables. After restoring the original value of `PATH` the call to `ls` works again.

Changing your search path comes in handy e.g. if you are writing your own programs or scripts and store them in a directory in your home directory – adding this directory to your search path will allow you to call your creations without typing their whole paths.

But apart from your search path, what else is in your environment? To find it out, use `printenv(1)`:

```
$ printenv
SHELL=/bin/bash
XTERM_SHELL=/bin/bash
EDITOR=/usr/bin/vim
PWD=/home/me/unixcourse
LOGNAME=me
TERMINAL=xterm
HOME=/home/me
LANG=en_US.UTF-8
TERM=xterm
USER=me
VISUAL=/usr/bin/vim
DISPLAY=:0
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

The output has been truncated to remove irrelevant/sensitive stuff. You can see that things like the path to your shell executable, the language of your system, and your username. Some of these variables are of vital importance to some programs:

PWD: If you ever wondered, how `pwd(1)` finds out in which directory you currently are, now you know the answer. It simply reads the `PWD` variable from your environment and prints it to the screen. Whenever you `cd` into some other directory, the shell updates this variable so a new call to `pwd` will again give the correct result.

HOME: You already know that if you type `cd` with no arguments, you will land in your home directory. The `cd` shell builtin knows where this is because it reads this variable. Note that the tilde `'~'` is also an alias for your home directory, i.e.

```
$ cd ~/unixcourse
```

will change to the directory `unixcourse` in your home directory.

SHELL: As has been mentioned several times, there are different shells with different capabilities. Reading this variable, other programs can tell whether the shell you use has certain kinds of features or not.

Changing Your Environment Persistently

All the changes we've done so far are only valid for the currently running shell. Just set your `PS1` variable to anything you want and then start a new shell:

```
$ echo \"$PS1\"
"$ "
$ PS1="\u@\h $ "
me@machine $ bash    # this starts a new bash
$ echo \"$PS1\"
"$ "
```

However, this hints at something else: Somehow the shell must know to what value it sets your PS1 variable initially.

This “somehow” are configuration files. For the bash shell there are two main configuration files in your home directory: `.bashrc` and `.bash_profile`³⁵. They serve different purposes:

`~/.bashrc`: This is the startup file which the shell reads everytime it is called as an interactive shell, i.e. for our purposes whenever you start a new shell by opening a new terminal emulator or by executing the `bash` command.

Things you set here are your PS1 variable or aliases.³⁶

`~/.bash_profile`: This is your personal initialization file which is executed for login shells, i.e. for our purposes this means it is read once when you login. Your `PATH` variable is usually set here.

You can customize both of these files to your heart’s desire. These files are actually shell scripts (which we will cover later), so you can do some scripting in there if you want.

4.8 Shell Scripting Basics

Until now, we typed one command at a time, sometimes combining them with pipes, but that was it. However, it is able to write script files which execute a sequence of commands. We’ll start with a very basic script – just open a file with your favourite editor and type in the following lines:

```
#!/usr/bin/env bash
```

```
echo "Hello, World!"
```

```
exit 0
```

Save it as e.g. `script1.sh`. Let’s analyse what each does:

³⁵Traditionally this was just `.profile`. You may still find it on your system and everything is fine as long as you don’t have both of them.

³⁶An shell alias is a way to create a shortcut for a command including its arguments, e.g. if you type (or put in your `.bashrc`) `alias ll="ls -l"`, you defined an alias called `ll` which is just `ls` with the `-l` option included.

Table 4.6:

<code>#!/usr/bin/env bash</code>	This line defines the interpreter for the script, which program should execute it. In our case, we want the bash shell to run it. In many scripts you'll see people writing <code>#!/bin/bash</code> instead of the above. This is ok as long as you use only systems where the bash is actually located in <code>/bin</code> . If it is installed in e.g. <code>/usr/bin</code> (which is the case in many BSDs) your script will fail. <code>#!/usr/bin/env bash</code> , on the other hand, checks your <code>PATH</code> variable for the location of the bash in your environment and executes it. As <code>/usr/bin/env</code> is a standard location across different kinds of UNIX-like operating systems, this syntax is more portable and that's why we prefer it.
<code>echo "Hello, World!"</code>	Ok, you understand that one by now ;)
<code>exit 0</code>	Return an exit code of 0 which usually indicates successful execution.

To execute a script in the current directory, you can't just type its name. Why? Because the current directory is (hopefully) not in your search path and therefore your shell doesn't look here for executables. Therefore, you have to prepend `./` to the script name³⁷. Let's try it out:

```
$ ./script1.sh
bash: ./script1.sh: Permission denied
```

Huh, what happened? Take a closer look at the script:

```
$ ls -l script.sh
-rw-r--r-- 1 me me 51 Aug 26 20:12 ./script1.sh
```

Can you spot the problem? If not, don't worry, we'll explain it: In 4.1 we talked about permissions. As you can see in the above output, there are no execute permissions on the script. To add them, we use `chmod(1)` which was discussed in 4.2:

```
$ chmod u+x script1.sh # add execution permission for the user
$ ./script1.sh
Hello, World!
```

Our next example will be a script which takes arguments and prints them to the screen:

³⁷Remember that `.` is the current directory, i.e. `./script1.sh` will be expanded to an absolute path.

```
#!/usr/bin/env bash

echo "Hi, I am $(basename $0)"
echo "You passed $# arguments to me. They were:"
for args in $@; do
    echo "  $arg"
done

exit 0
```

Before we explain the details, let's make it executable and run it:

```
$ chmod u+x script2.sh
$ ./script2.sh
Hi, I am script2.sh
You passed 0 arguments to me. They were:
$ ./script2.sh here are some arguments
Hi, I am script2.sh
You passed 4 arguments to me. They were:
  here
  are
  some
  arguments
```

And now, the lines you may not understand yet in detail:

Table 4.7:

<code>echo "Hi, I am \$(basename \$0)"</code>	That's a heavy weight! First, the <code>\$(...)</code> expression is a so called <i>command substitution</i> . This means that the expression will be substituted by whatever the command inside the parentheses printed to stdout. <code>basename(1)</code> is a command which prints the last element of a path, e.g. <code>basename /this/is/a/path</code> will print "path". Finally, <code>\$0</code> is the "zeroth" argument you passed to the script, which is set to the name by which you called, i.e. in our case <code>\$0</code> will expand to <code>./script2.sh</code> . Thus, if you go through this step by step, first "basename \$0" is expanded to "basename ./script2.sh" which prints "script2.sh" which is printed by echo.
<code>\$#</code>	This is a special variable which contains the number of arguments that were passed to a script, excluding <code>\$0</code> .
<code>\$@</code>	This is a special variable which expands to all the arguments passed to the script, separated by whitespace.

And finally, we have a for-loop. This kind of loop repeats a certain number of times, i.e. in the above example it is executed as many times as there are arguments to the script. The basic syntax of a for-loop in shell scripts will be given below.

While our two example scripts didn't do anything useful, scripting in general is very powerful and enables you to work efficiently. Sysadmins especially love scripts to get summaries of the system state from different tools. Unfortunately, there is no time in this course to cover them, but if you start using the command line regularly, you'll soon find a lot of use cases for scripts.

4.8.1 Syntax of Common Constructs

This section provides you with a quick overview of the commonly used programming constructs in shell scripting.

if-Statements

An if-statement executes some code based on whether a condition is true or false. It's basic syntax is:

```
if CONDITION; then
    STATEMENT(S)
[ elif CONDITION; then
    STATEMENT(S)
]...
[ else
    STATEMENT(S)
]
fi
```

The `elif` and `else` blocks are optional, indicated by the brackets.

for-Loop

You've already seen the for-loop in action in our second script above; it repeats one or more statements for a number of times. Syntax:

```
for VAR in LIST; do
    STATEMENT(S)
done
```

For those who programmed in C-style languages: `bash` also supports the following syntax:

```
for (( expr1 ; expr2; expr3 )); do
    STATEMENT(S)
done
```

while- and until-Loop

In contrast to a for-loop, a while-loop repeats something as long as a condition is true:

```
while CONDITION; do
    STATEMENT(S)
done
```

The shell also offers an until-loop, which repeatedly executes some code *until* a condition is true:


```
until CONDITION; do
    STATEMENT(S)
done
```

4.9 Installing Programs With and Without Rootly Powers

In general, there are two ways to install programs on a (GNU/)Linux system: Either through a package manager or from source.

4.9.1 Package Management

(GNU/)Linux systems usually want you to install programs using a packet management system. Today, you are most certainly familiar with this concept if you are using Google Play on Android or Apple's App Store on iOS. However, this concept of a centralized location from which you download and install everything was pioneered by (GNU/)Linux distribution in the 1990ies. The good thing about such repositories is that they are managed by knowledgeable people who take care of updating packages in the repositories and make sure that everything works together. Before package management was invented you had to do this all by yourself!

The utility for package management on Debian GNU/Linux and distros derived from it (e.g. Ubuntu, Linux Mint) is `apt(1)`, the “Advanced Package Tool.” `apt` doesn't just install packages, it also takes care of dependencies, i.e. if the package you want to install needs another package to work properly, this other package will be installed by `apt` as well which makes it very convenient to use.

`apt` also keeps track of all the files a package installs on the system which makes it easy to cleanly remove everything if you decide to uninstall a package at a later point.

Additionally, `apt` is also used to update all the packages installed on your system.

The basic options for `apt` are:

`install PACKAGE...` Install one or more packages.

`remove PACKAGE...` Remove one or more packages, but leave their configuration files on the system.

`purge PACKAGE...` Remove one or more packages and also delete configuration files.

`update` Download package information from all configured repositories. This doesn't install any updated packages, it just updates the information about which packages are available in which version.

`upgrade` Install all available updates for packages that are already installed on your system. Usually, you first do `apt update` to get information about new packages followed by `apt upgrade` to actually apply the updates.

`list PATTERN...` List all packages which match one of the given patterns and indicate whether they are installed or not. Wild cards are supported.

`show PATTERN...` Similar to `list`, but shows a lot of additional information about matching packages like who is the maintainer of the package, a description what the package is, dependencies, and much more.

Note that installing and removing packages with a packet manager always requires root privileges. To elevate your privileges use `sudo(8)`, e.g.:

```
$ sudo apt update && sudo apt upgrade
```

The “&&” operator executes the second command only if the first one succeeds. You will be prompted for your password³⁸ and if you are a member of the `sudo` group (which you should, if you followed the installation instructions carefully) your system will be updated.

4.9.2 Installing Programs from Source

While package management is great, there are some situations in which you can’t use it, e.g.

- You don’t have root privileges and your sysadmin doesn’t want to install the program you need.
- The version in the distro’s repository is outdated. Many distros only allow well tested version in their distros, which can introduce a significant delay until newer versions are added.
- You want to install a program which is not available as a package. This is e.g. often the case if you are a developer and want to install some cool new tool you have written yourself.

In these situations, you have to grab a copy of the source code, compile it and then install the executables to the right places on your system. While this may sound complicated (and most certainly is in some situations), in most circumstances it is pretty simple to do this:

1. Download the tar-Archive of the source code.
2. Extract it and `cd` into the new directory.
3. Read the `INSTALL` file and check if you have to pass any options/arguments to the following commands.
4. Prepare the compilation by calling

```
$ ./configure
```

³⁸Nota bene: The shell doesn’t give any visual hints while you type your password, i.e. also no “*” for each character. Just type in your password and press ENTER, it will work ;)

5. Compile by calling

```
$ make
```

6. (optional and only possible if you have root privileges) Install the executables to the standard locations of the system like `/usr/local/bin` which are usually included in users' search paths:

```
$ sudo make install
```

You can also combine the last three steps in one line:

```
$ ./configure && make && sudo make install
```

Et voilà, you compiled a program from source code and installed it on your system.

This approach enables you to install everything, but it has some important caveats:

- This method can't handle dependencies, i.e. if you need another program, the installation process will fail and you have to install the other program first.
- Unless you do it yourself, there is no easily accessible documentation about the location(s) to which files were installed, i.e. removing stuff later can be really tedious.
- You have to check the code for yourself as there are no maintainers who already ensured that it will work on your system and is non-malicious.

4.9.3 Your Very First Compiled Program

Now that you've got a very small taste of bash scripting as well as compiling programs from source you may wonder what exactly "compiling" actually means. This denotes the process of translating human-readable source code into so-called machine code which is very fast for a machine to interpret but *very* hard for a human to read and understand. A *compiler* is a program that does this translation and you'll work a lot with them in your upcoming studies. To give you a very basic understanding of what the process entails, fire up a text-editor and write the following C++ program:

```
#include <iostream>

using namespace std;

int main() {
    cout << "Hello, world!" << endl;
    return 0;
}
```

Save it as `first.cpp`. Pretty easy, no? The small shell script we encountered earlier is not compiled, but *interpreted*. This means that it is read line by line by your shell and each line is interpreted as if you would type it into your terminal window.³⁹ For this reason, we just had to change the permissions on the file to make it executable. For our new shiny C++ program above, this approach won't work – try it out if you don't believe a PDF someone handed to you.

In order to make this executable, we have to compile it first. For such a simple program, this is very easy to do:⁴⁰

```
$ g++ -Wall -Wextra -O0 -std=c++23 -o first first.cpp
$ ./first
Hello, World!
```

So, `g++` is the C++ compiler on our Debian system and apart from our program we provided some other options:

`-Wall`: Show all warnings.

`-Wextra`: The last option was a lie – show even *more* warnings.

`-O0`: Don't use optimizations. Compilers use all kind of neat tricks to translate your source code into efficient machine code. This is great for everyday usage but bad for learning because when you're learning you want the compiler to keep all your mistakes to learn from them.

`-std=c++23`: Use the C++ Standard from 2023. Languages such as C++ are standardized and some things change between one standard to the next, so make sure to check if you have to change any of your programming behaviour *before* switching standards.

`-o first`: Write the compiled program to a file called `first`. If you don't provide this option, `g++` will call the output file `a.out`.

And voilà, your first compiled program \o/

³⁹That is *almost* true, and we'll leave it at that for this course ;)

⁴⁰Yes, `g++` uses a single dash for longer options, which is a bit confusing...

5 Final Words

This 3-day workshop barely scratched the surface of (GNU/)Linux systems and the power of the command line. For example, we didn't do any networking stuff, we didn't do system administration, we didn't mess with hardware or mounted and unmounted disks, etc.

There is a *lot* to learn about the topics we covered in this introductory workshop and we hope we could incite your interest and made you hungry for more. As mentioned in the , there are lots of sources, online and offline to help you get more familiar with the command line and the concepts behind it.

If you are looking for a resource that goes beyond this course but still starts with the basics I (Dimitri Robl) can whole-heartedly recommend William Shott's "The Linux Command Line" about which you can find more information [here](#). In its about 500 pages it covers all the basics of command line wizardry in a fun way. I started learning the command line with it and loved every page :)