# P2T: C Programming under Linux

## Lab 1: Introduction to the command line and the C programming language

## Introduction

This lab will serve as a basic introduction to the command line using Bash – the Bourne Again Shell. A shell is a program which interprets your commands and lets you interact with the operating system. It can be used interactively, where you type commands in one at a time, or you can provide a set of commands in the form of a script. Once you have completed this lab, you should be relatively comfortable using the command line interactively to navigate the Linux file system and work with files.

We will also start learning the C programming language with two activities, involving simple programs to introduce you to the basics of the language and the "compilation" process.

The assessed work in this lab consists of questions and programming activities which will be marked by your demonstrator. Each question states the number of marks available.

## Getting Started

Start by opening the **Terminal** application. If you are using XFCE, the desktop environment that we recommend for this course, you will find this on the launch panel at the bottom of the screen. The Terminal application will give you access to a Bash shell in which you can enter commands.

When using the command line, it is important to keep track of which directory you are currently working in. Terminal should start in your home directory, and you can check this by typing **`pwd`** ("print working directory") and pressing **`Enter`**. The path to the current directory—also known as the working directory—will be displayed, which should be something like **`/home/0123456a`**. If it's not, speak to a demonstrator!

If you want to know what files and folders are present in the current directory, you can list its contents using the **`ls`** command (with a lowercase letter 'l').

The files needed for this lab can be obtained using a tool called **`git`**. We will explore git later in the course, but for now simply enter the following command:

```
git clone https://bitbucket.org/glaphysp2t/linux-lab01.git
```

This will create a copy of the files for this lab in a directory called **`linux-lab01`** within your home directory. Confirm you have the necessary files by listing the contents of this new directory: **`ls linux-lab01`**

## Note: Learning about Linux Commands

You can find more information about Linux commands by reading their manual pages. Some of the questions in these labs will require you to find information in the manual pages, so it is important that you get used to using them.

You can access a manual page by using the command **`man COMMAND`**, where **`COMMAND`** is replaced with the name of the command you want to find out about. Try it now with the **`ls`** command:

**`man ls`**

The manual page provides a synopsis which tells you what arguments the command takes, along with a detailed description of what these arguments do. Depending on the command, you may find other information, such as the status codes returned by the command when it finishes, or examples of usage. You can use the arrow keys or Page Up and Page Down to move around within the manual page. When you're finished, press **`q`** to quit.

You can search for man pages using the **`apropos`** command. **`apropos delete`** will search for manual pages which have a description containing the word "delete".

Manual pages exist for more than just commands. For example, **`man intro`** contains a basic introduction to user commands.

As more commands are introduced, you should read their manual pages to get an idea of what the command can do. Reading the manual pages and trying out the various options is by far the most effective way to learn Bash.

As well as the manual pages, many commands will display a summary of their main options when given the **`--help`** argument, for example: **`ls --help`**

A summary of the main commands introduced in this lab is included at the end of this document. In addition, you will find a copy of **`Commands.pdf`** among the files you downloaded in the **`linux-lab01`** directory. This contains a list of some common commands and their most useful options. Try some out before you continue.

If you are stuck, please remember to ask the demonstrators, who will be happy to help during the labs. You are of course allowed to refer to lecture notes, books and the Web while completing these tasks.

# Files and Directories

You are going to create a basic directory structure which could be used to store the data and documents for an experiment. Some large experiments result in hundreds of even thousands of files being created – sorting and storing them in an organised manner is essential.

1. Use the **`pwd`** (print working directory) command to check that you are in your home directory. This is usually the directory you start in when you open the terminal, but if you find you are in another directory, the command **`cd`** (change

directory) without any arguments will take you to your home directory.

2. Create a directory called **experiment**. If you don't know how to do this, you can look up the manual page: **man mkdir**. Once created, you should see the new directory when you list the contents of your home directory using the **ls** command.
3. Change your working directory to the **experiment** directory using the **cd** command.
4. Create three new directories within the **experiment** directory called **docs**, **data** and **results**. You should now have a directory structure like this:

   **experiment**
   **experiment/data**
   **experiment/docs**
   **experiment/results**

   You can check that you have the correct structure by using the command **ls -R** to list the contents of the directories recursively.
5. Use the **cd** command to return to your home directory, and then change into the **linux-lab01** directory which was created when you ran the **git clone** command at the start of this lab.
6. You are going to run a Bash script which will generate some sample data files. Use **ls** to check that the script called **generate.sh** exists. Because this script is not stored in one of the locations Linux looks to find commands, we need to use the command **./generate.sh** to run it. Do this now; it may take several seconds to run.
7. Without changing directory, check the contents of the **experiment** directory using the command **ls ~/experiment**. Here, the tilde character (**~**) will be automatically substituted with the absolute path of your home directory. The **experiment** directory should now contain fifty **.data** files.
8. Change back to the **experiment** directory. You need to move the data files into the **data** directory. Start typing **mv 2**, but do not run the command yet. Instead, press the **tab** key twice. You will see a list of all the files in the directory beginning with the character **2**. Type **4**, and press **tab** again. This will complete the filename (when there is only one match, Linux fills in the value rather than displaying a list of possibilities).

   Do the same again to add the name of the destination directory. The complete command should now be:

   **mv 24.data data/**

   Run this command, and check that the file has been moved to the **data** directory.
9. Moving each file individually like this would take a long time. There's a much quicker way to select all the **.data** files by using the * character. Run the

command **mv *.data data/**. Here, * will match any number of characters (including none at all) meaning that all files with a name ending **.data** will be moved.

Change into the **data** directory and use **ls** to verify that all the files have now been moved.

10. Let's say you want to filter out some specific data files to examine in more detail later. Create a new directory called **5s**, and use the **cp** command to copy any file which has a name containing the character **5** into it.

    *Remember you can use the * character to match multiple files at the same time.*

11. Imagine you wish to perform multiple experiments as part of a larger project, all creating multiple data files. Create a new directory in your home directory called **project** to store all this data.

12. Copy the experiment directory into the **project** directory, renaming it **exp1** in the process. You will need to use the **-R** argument to copy a directory.

    You should be able to do this using a single **cp** command. Keep a note of the command you used.

13. Try to remove the old experiment directory using the **rmdir** command.

14. Use the **rm** command to remove the old **experiment** directory. Check its manual page or refer to **Commands.pdf** to see how to delete a directory and its contents.

## Questions

These questions should be answered in the file **answers.txt** which can be found in the **linux-lab01** directory created at the start of this lab. Please talk to your demonstrator if you find any of the questions tricky.

1. From your home directory, what is the output of the command **ls -R project**? (Copy what is displayed in the terminal into the answers file.) **(1)**
2. What command did you use to copy and rename the directory in step 12? **(1)**
3. Why did **rmdir** give an error in step 13? **(1)**
4. What command did you use to delete the directory in step 14? **(1)**
5. Read the man pages for **ls**.

   a) What does **ls -a** do? **(1)**

   b) What is special about the name of a hidden file? **(1)**

c) What is the name of the hidden file in the **~/project/exp1/data** directory? **(1)**

6. Every directory contains hidden directories named **.** and **..**

a) What do these two directories refer to? **(2)**

b) What command would you use to list all files and directories, including hidden ones, except **.** and **..**? **(1)**

7a) How would you display a list of files sorted by size? **(1)**

b) How would you display a list of files sorted by modification time? **(1)**

8. How would you use **cd** to change from **~/project/exp1/data** to **~/project**:

a) Using relative paths? **(1)**

b) Using absolute paths? (Note that while ~ is substituted with the absolute path to your home directory, ~ itself is not an absolute path but rather a convenient shortcut.) **(1)**

9. Open **generate.sh** using the command **less**.

a) How can you tell **less** to display line numbers? **(1)**

b) What is line 21 of **generate.sh**? **(1)**

10. Read the manual pages for the **head**, **tail** and **wc** commands.

a) How many lines do **head** and **tail** show by default? **(1)**

b) How can you make **head** display five lines? **(1)**

c) Which argument makes **wc** show only the number of lines in a file? **(1)**

***(19)***

## Note: Saving Time

As described earlier in this lab, using the **tab** key to complete the names of commands, files and directories automatically can greatly reduce the amount of typing you need to do!

The Linux terminal has several other time-saving features. You can use the up and down arrow keys to revisit commands you have run previously, and the left and right arrow keys to make changes in the middle of these commands. If you've just entered a long command which didn't quite work, you don't need to start typing it out from the beginning again.

You can also use the **CTRL + R** key combination to search through your command history and repeat a command. Press **CTRL + R**, and start typing part of the command you are looking for; Linux will display matching commands from your history, starting with the most recent. Press **CTRL + R** repeatedly to scroll through earlier commands. Once you've found the command you are looking for, use the arrow keys to make changes or press **Enter** to run it.

# File Permissions

Most Linux file systems have a simple but powerful set of file permissions that control who can do what with particular files and directories. Being able to manage these permissions is extremely important.

1. Change to the **linux-lab01** directory and use the **touch** command to create a new file called **important**.
2. Use the long listing format for the **ls** command to display file permissions and other information. (Check the manual pages if you are unsure how to do this.)

   You should see a line similar to the following:

   ```
   -rw-r--r--  1 james james 0 Jan 31 15:50  important
   ```

   We're mostly interested in the first part: **-rw-r--r--**. This tells us that the file is readable by everyone, but only its owner (in this case, the user called **james**) can write to it.

   ***Refer to Linux Lecture 2 for information on how to read file permissions.***

3. Since this file is important, you are going to protect it by making it read-only. You can do this using the **chmod** command. Run the command **chmod a-w important**. This will modify the permissions for all users (**a**) by removing their write permissions (**w**).
4. Open **important** in a text editor, add some text, and try to save the file. You won't be able to. Now, exit the text editor and try to delete the file with **rm important**. You will be asked to confirm the deletion, giving you the chance to back out before the file is lost forever.

File permissions also determine whether or not a file that contains a program or script may be executed. In the interests of security, files are not executable by default. You will need to know how to make files executable before you are able to run the Bash scripts which you will write in lab 3.

5. In the **linux-lab01** directory there is a C program called **secret** which contains the answer to one of the questions at the end of this section. Try running this program:

   ```
   ./secret
   ```

   Linux will refuse to run the program, giving you a "permission denied" error because the file is not executable.

6. Use **chmod** to make the program executable by using the **+x** argument. Run the program again, and keep a note of its output for later.

# Questions

11. A file has been created with the following permissions: **-rw-rw---x**

    a) Describe who can do what with the file. **(3)**

    b) Which command would you use to change the permissions so that everyone can read the file, but only the owner can write to or execute it? **(2)**

12. By looking at the file permissions, how can you tell if a file is a directory? **(1)**

13. Read the man pages for **rm**.

    a) How can you force **rm** to ask for confirmation before deleting a file? **(1)**

    b) How can you stop **rm** from asking for confirmation before deleting a write-protected file? **(1)**

14. In step 6, what was the secret message displayed when you ran **secret**? **(1)**

***(9)***

# C Programming – Compiling, Bug Fixing, Scope and Lifetime

## Activity 1

The code below can also be found at **~/examples/p2t/lab01/ex1-orig.c**

```
int k = 4;
int main(void) {
      int i = 50;
      unsigned int j = i * 2;
      double k = 1.0
      {
            float i = 5.0;
            printf("The value of i is %6.1f\n", i);
            k = i * j;
            i *= 6;
       }
      double j;
      i = k + i; //or i += k
      printf("Now , the value of i is %d\n", i);
      return 0;
}
```

15. Make a new directory in your home directory called **clab01** and use **cd** to change into it.

Make a copy of the code above in the new directory.

You can try to compile this C source code into an executable program with:

```
clang ex1-orig.c
```

Try it, and notice that **clang** fails, and gives you a number of error messages and warnings as a result.

a) Fix the first error by following the advice that **clang** gives you (*add a semicolon to the end of line 5*) and save the file again.

By repeatedly trying to compile the file, and then following the advice from **clang**, fix the other errors or warnings. Add comments to the code each time to say what you did. **(2)**

Now that the code compiles, you should have a new file – called **a.out** – in the same directory. (Use **ls** to list your directory's contents to check.)

16. Read through the C source code. What do you expect the program to print?

Run the code by typing:

```
./a.out
```

in the terminal.

a) Does it do what you expect? Add comments to the code describing what it does (and why). **(2)**

17. There are two different variables called **k** and two different variables called **i** in this code.
One **k** is in the "file scope" – it will be the **k** that we refer to unless another **k** is declared in a deeper scope, and its lifetime (the period when its data is kept) is the entire program duration.
The second **k** is declared in a deeper scope—the interior of the main function—and thus it is this **k** that we are referring to for all of the rest of the main function, including its inner block. This "block scope **k**" is only kept for the duration of the main function's block and is lost when it ends.

a) Write comments in the source code explaining the scope and lifetime of the two variables called **i**. **(4)**

b) Add two **printf** statements to the code, printing out the value of each variable called **k**, at the point at which that variable is changed for the last time. Remember, statements can only be written inside a function, so for the block-scope **k** you need to find a place inside **main** where you can refer to it. **(2)**

<u>The resulting (source code) file, with comments, is your submission.</u>

**(10)**

# C Programming – Your First Program

## Activity 2

If $z_0 \equiv x_0 + y_0 i$ and $z_1 \equiv x_1 + y_1 i$ are complex numbers, then their quotient, $\frac{z_0}{z_1}$, can also be written in real and imaginary parts as:

$$\Re\left(\frac{z_0}{z_1}\right) = \frac{x_0 x_1 + y_0 y_1}{x_1^2 + y_1^2}$$

$$\Im\left(\frac{z_0}{z_1}\right) = \frac{x_1 y_0 - x_0 y_1}{x_1^2 + y_1^2}$$

We will write a C program to calculate the real and imaginary parts of a quotient of complex numbers, and print it to the terminal.

18. Using the **`touch`** command (which creates an empty file with the name specified), create a file called **`exe2.c`** in the **`clab01`** directory:

    **`touch exe2.c`**

    Open this file in the text editor of your choice.

a) Write the framework of a basic C program (an **`#include`** statement that lets you use **`printf`**, and a basic **`main`** function that does nothing except return 0). **(1)**

    Check your code compiles using **`clang`** (and fix any errors before you continue).

19. Inside the **`main`** function:

a) Declare four variables (**`x0`**, **`x1`**, **`y0`**, **`y1`**) as integers and assign values to them. These are your real and imaginary parts of $z_0$ and $z_1$. **(1)**

b) Declare two variables (**`re_quot`** and **`im_quot`**) as double-precision floating points.

c) Using appropriate mathematical operators, and potentially a suitable *cast*, assign the results of the equations above to the two variables. You may use additional variables if you wish. **(2)**

d) Write an appropriate **`printf`** statement that prints out the calculation done, and its result. For example, this might look like: **(2)**

    **`The result of 2 + 3i divided by 1 + 1i is 2.5 + 0.5i.`**

You should print floating point values to 1 decimal place.

20. Appropriately comment and lay out your code, including a note on why the casts are needed if you used them. **(3)**

21. Your code should compile correctly with no errors or warnings. (It's often useful to try compiling your code whilst writing it to see what errors **`clang`** picks up as you go along. In this case, also try testing you get the right values for different complex numbers.) **(1)**

The source code file, with suitable comments, is your submission for this exercise

***(10)***

## Note: Clang options

In this lab, we have used the Clang compiler to compile your source code to a program called **`a.out`**:

```
clang mysourcecode.c
```

If you want to call the resulting program something else, you can give clang an output name using the **`-o`** ("o" for output) option, followed by the name to use:

```
clang mysourcecode.c -o myprogram
```

This creates a file called **`myprogram`**.

There are many other options for clang, some of which we will look at in later labs.

## Submitting Your Work

### Note: Making an Archive of your Source code to submit Exercises.

There are many tools available to make an archive file containing your source code and other files in a single file. We recommend that you use one of these to allow you to submit a single file for each Lab.

### Tar

The tar program was originally invented to allow files to be packaged together for archiving on magnetic tape (hence "tar" – "**t**ape **ar**chive"). It is still useful today for packaging multiple files together in one place, but its syntax is a little complicated.

To create a tar archive called **`lab1.tar`** containing multiple individual files, type (at a Bash shell or other terminal):

```
tar cvf lab1.tar myfile.c myotherfile.c
```

(Assuming the files you want are **`myfile.c`** and **`myotherfile.c`** and are in the same directory as you—you will need to specify their path otherwise.) You can list as many files as you want for inclusion—remember, the first filename is the name of the archive you're making, not a file to be added to it!

You can also tar up a whole directory:

```
tar cvf lab1.tar mydir
```

You can get tar to list what's in a tar file with:

```
tar tvf lab1.tar
```

And you can extract a tar file again with:

```
tar xvf lab1.tar
```

### Zip

Zip is a popular compressed archive creator, available on most platforms.

You can make a zip archive called "lab1.zip" from individual files with:

```
zip lab1.zip myfile.c myotherfile.c
```

(with assumptions as above for tar)

If you want to zip an entire directory, you need to specify the –r option (for "recursive"):

```
zip –r lab1.zip mydir
```

*If you don't specify* ***–r****, zip will make a file containing just the name of the directory, and nothing inside it.*

To unzip a zip file:

**`unzip lab1.zip`**

To just check the files inside a zipfile (without uncompressing it)

**`unzip –l lab1.zip`**

Your work should be submitted as a suitable archive file containing:

- **`answers.txt`** which includes your answers to the first part of the lab (you may submit this inside the directory below).
- the **`clab01`** directory, containing two C source code files, one for each activity.

The archive should be submitted through Moodle. You do not need to submit any of the other files you have modified.

## Summary

Following this lab, you should know:

- The difference between an absolute and relative path, and how to navigate the Linux file system using the **`cd`** command.
- How to view the contents of a directory using the **`ls`** command.
- How to copy, move and delete files and directories using **`cp`**, **`mv`** and **`rm`**.
- What the various file permissions mean, and how to change these to make a file executable.
- How to recognise hidden files, and the meaning of the special hidden directories **`.`** and **`..`**
- How to display the start or end of a file, and how to find out how many words or lines it contains.
- How to use Linux manual pages to learn about a command you are unfamiliar with.
- How to write a simple C program and compile it with **`clang`**.
- How to use the errors from a compiler to help fix bugs in your C code.
- What the scope, lifetime, and type of a variable are in C.

A list of some of the most important commands introduced in this lab is provided overleaf.

## Commands used in this Lab

| Command | Summary | Description |
|---|---|---|
| **cd** | Change directory | Change to your home directory |
| **cd PATH** | Change directory | Change to **PATH** |
| **chmod MODE FILE** | Change file mode | Change the permissions on **FILE** to **MODE** |
| **clang SOURCE** | Compile source code | Compile the code in **SOURCE** to an executable |
| **cp SOURCE TARGET** | Copy | Copy **SOURCE** to **TARGET** |
| **head FILE** | Display first part of file | Display first lines of **FILE** |
| **less FILE** | Display file contents | Display the contents of **FILE** |
| **ls** | List directory | Display the contents of the current directory |
| **ls PATH** | List directory | Display the contents of **PATH** |
| **ls -R** | List directory | Display the contents of the current directory and all subdirectories contained within |
| **man COMMAND** | System reference manual | Display the manual page for **COMMAND** |
| **mkdir PATH** | Make directory | Create the directory **PATH** |
| **mv SOURCE TARGET** | Move (or rename) | Move **SOURCE** to **TARGET** |
| **pwd** | Print working directory | Display the path of the current directory |
| **rm FILE** | Remove | Remove **FILE** |
| **rmdir DIR** | Remove directory | Remove directory **DIR** if empty |
| **tail FILE** | Display last part of file | Display last lines of **FILE** |
| **touch FILE** | Update file timestamps | Update the access and modification time of **FILE** to the current time (an empty file is created if **FILE** does not already exist) |
| **wc FILE** | Word count | Displays the number of lines, words and bytes in **FILE** |