

Tutorial 0

First steps with Linux and C

Peter Košovan*, Marcello Sega†

October 26, 2011
ICP, Uni Stuttgart

1 Checklist: what you need to know for the next tutorial

- Basic file and directory operations: create, copy, rename, move, delete
- Edit a text file item Remote login via gate using ssh
- Read and understand elementary code in C
- Create a plot from data in an ASCII file

2 Introduction to Unix/Linux

2.1 Introduction

Most Computational Physics applications run under Unix or Unix-like systems. One example of a Unix-like system is Linux, which runs on computers at ICP. The purpose of this tutorial is to give a short introduction to using this operating system and environment, to enable its basic use to those, who have (almost) never been using it before. Linux is an Open Source software, it can be downloaded for free and if you have not installed it yet on your own computer, it is worth trying out.

If you are new to Linux, we recommend to read a bit more about Unix, Linux and related topics. Some links are provided on the tutorial web-page. The basic concepts of Unix-like systems have not changed too much since its introduction in 1960s, therefore also an older book may well serve the purpose, even though some information may be outdated.

*kosovan@icp.uni-stuttgart.de

†sega@icp.uni-stuttgart.de

2.2 Directory structure

Table 1: Common directories in a Linux system

/	System root directory
/home/<user>	Home directories of all users (writable for the user, usually readable for everyone)
/tmp	Directory for temporary files (writable for everyone)
/etc	System configuration settings (writable for privileged users only)
/usr	User applications (writable for privileged users only: most programs are found in /usr/bin)
/lib	System libraries (writable for privileged users only)
/mnt	Usual mount point of other physical drives

There are certain standard directories under all Linux systems. Their names and typical contents are listed in Table 1. Those used to Windows should note that the directory separator is a slash, "/", not a backslash, "\". All filesystems are mounted as directories. There are no C, D, ... drives. Therefore, the user does not have to care about where his data are physically stored, he always accesses them in the same way as any other local directory.

Sometimes, it may be useful to know where your data are physically stored. For example, in our CIP Pool, /home directory is on a remote network drive which makes your personal directory available from all CIP Pool computers. The network drive is slower than local drives. Therefore if your program produces bigger amounts of data, it may be better to work in the /tmp directory which mounted on a local drive. But be careful: when the work is done, move all important data to your /home. The /tmp is dedicated for temporary files and they may be deleted later by the administrator.

If you are interested on which devices (disks, memory sticks, RAM, remote computers) a branch of the directory tree is mounted, have a look at the `df` or `mount` commands.

2.3 Basic commands

A selection of most commonly used Linux commands is listed in Table 2. A more extensive list you can find in the Linux cheat sheet provided on the tutorial website. All commands have a number of different arguments and switches to alter their behaviour. Nobody can remember all of them, therefore there are manuals (man pages) which can be accessed by typing

```
man <command_name>
```

This gives you a full documentation of the command, including all options and peculiarities. To quit a man page, hit "q". If you don't know yet what is the command related

Table 2: List of most common commands

command	argument type	what it does
<code>ls</code>	<i>file or directory</i>	list directory contents
<code>cd</code>	<i>directory</i>	change directory
<code>mkdir</code>	<i>directory</i>	create a directory
<code>rm</code>	<i>file</i>	remove a file
<code>mv</code>	<i>file</i>	move a file
<code>cp</code>	<i>file</i>	copy a file
<code>wc</code>	<i>file</i>	count lines, words and characters in file
<code>cat</code>	<i>file</i>	print file contents to standard output
<code>more</code>	<i>file</i>	view file contents interactively
<code>less</code>	<i>file</i>	similar to <code>more</code>
<code>echo</code>	<i>string</i>	print the <i>string</i>
<code>grep</code>	<i>string file</i>	print lines from <i>file</i> which contain <i>string</i>
<code>history</code>		print commands you have recently executed
<code>bg</code>		put a job in the background
<code>fg</code>		put a job from background to the foreground
<code>jobs</code>		list the jobs launched from your actual shell

to a given subject, try the `apropos` command, like in

```
apropos compiler
```

Every command listed in Table 2 is a small program and it is executed in the same way as any other program. The only difference between a system command and other program is, that system commands are found on all Unix systems.

2.4 Users, rights and privileges

Linux, and all Unix-like systems, are by construction multi user systems. Therefore, each file and directory has a set of permissions which define, who is allowed to read, write and execute it. Each file has its owner and a group and the permissions for owner, group and all can be set separately.

The most privileged user is called “root”. Password to login as root is usually known only to one or very few people – system administrators. The superuser, as root is often called, is allowed to do any operation on the system.

In contrast, most common users are only have *read* permissions in system directories. Each user is the owner of his `/home/[username]` directory and has full access permissions to it. By default if you create a file in your `/home/[username]` directory you are given both *read* and *write* permissions, while others may read it but are not allowed to change it. As an owner, you may change access permissions to all your files.

Table 3: List of some special characters for the shell

character	name	meaning
	pipe	forward output from one program to the other
>	greater than	redirect the output to a file
<	less than	redirect the output from a file
&	ampersand	execute command in the background
!	exclamation mark	execute a command from history
.	dot	current directory
..	dot	directory one level above the current one
~	tilde	refers to the home directory of a user
*	asterisk	substitute an arbitrary combination of characters
?	question mark	substitute one character

2.5 Bash: an example of a shell

A shell is an interface for executing commands. There are many shells around and user is free to choose his favourite. One of the most widespread and preferred by ourselves, is **bash** which stands for Bourne again shell.

Shell interprets some special characters from a command line, to provide possibility of extended interactions with the system or between the programs you are executing. Some of them are listed in Table 3.

Here are some examples:

- `ls / > list` List the contents of the root directory and put the output in file `list` (which gets overwritten, if existed before)
- `cp ./* /tmp` Copy all the files (*) from the directory where you are now (.) into the `/tmp` directory.
- `ls /usr/bin/*cc*` List all files in `/usr/bin/` containing the string "cc".
- `~/myprogram < input | grep error` Feed program `myprogram`, located in your home directory (`~`) with the contents of file `input` (instead of typing them by hand) and search for the string `error` in the output.
- `./slow_program > output &` Launch `slow_program` in background and redirect the output to file `output`.
- `./program 2> /dev/null 1> log` Run `program` and redirect the output messages (`stdout`) to file `log` and the error messages (`stderr`) to the special file `/dev/null` (i.e., discard them). `stdout` and `stderr` are two special files (in Unix everything is a file!) that are used to output to the screen. Output sent to them can be redirected using the special `1>` and `2>` shell commands. The only difference between `stdout` and `stderr` is that the first is buffered, while the second is not.
- `!127` Repeat the 127th command in your `history`.

When executing a program, the complete path to it has to be given, beginning with the root directory, "/", e.g. `/bin/ls`. An alternative is to use a path relative to the working directory, e.g. `../programs/myprogram`. If you do not give a path, the shell searches for the program in directories defined in the environment variable `PATH`. Try typing `echo $PATH` to see them.

2.6 Tips and tricks

In the final section, we list several useful features provided by most modern Linux systems, which make our life much easier and help prevent typing mistakes.

- **Tab completion:** when you start typing a command and hit a tab key after you have typed first few letters, the shell completes the rest, provided there is one unique possibility. If there are more possibilities and you hit the tab twice, it displays all of them.
- **Copy-paste using mouse:** use a mouse to select a piece of text by the cursor. It is automatically stored in a buffer and when you click the middle button, it is pasted in the current cursor position.
- **Command-line history:** Suppose you executed a very complicated and long command, and you want to execute it again, perhaps with a small change. By hitting `↑` and `↓` keys, you can browse the history of commands you executed recently. If your command has been executed a long time ago, call a command `history` to get the whole history of your commands on your screen.

2.7 Suggested tasks

- In your home directory, create a directory named `tutorial_0`
- Change to `/tmp` and create a directory named with your user name.
- At the end of the tutorial, backup files which you might want to use later to your home directory.
- Try out the tips and tricks and playing around with the commands.

3 Useful programs

In this section, we describe some programs, which you will need for everyday work. You are free to use any program you like, but the ones listed below are known to work well for our purpose and are widely used in computational science community. They are all freely available for download and run under a number of operating systems, including Windows. They are also parts of all major Linux distributions.

3.1 Remote login and ssh

To connect to the CIP pool computers, use `ssh` which stands for secure shell. When using `ssh`, all communication through network is encrypted to prevent others from reading it. First you have to login to `gate` which is the only computer which allows for login from outside ICP: Type in the following, substituting your own username for `<username>`:

```
ssh <username>@gate.cip.ica.uni-stuttgart.de
```

After typing in your password you obtain access to shell command-line as if you were sitting directly at the machine. Specifically at CIP-pool, when you login to `gate` you find out that many programs are not available, because it indeed works just as a “gate” for remote login. To do some work, you have to use `ssh` again and login to one of the computers, e. g.

```
ssh palisander
```

to log in to `palisander`. For more information and names of other computers, should read the message you get when logging in to `gate`. When you do not specify the username, your current user name is used. When logging in to a computer within the same domain name, specifying computer name is enough.

3.2 Putty

`putty` is a tiny program for M\$ Windows which provides ssh connection. Running it is a good choice if you have a Windows PC and yet want to work on your homework from home.

3.3 Text editors: vi, emacs and others

Table 4: List of common text editors

editor	short description
vi	a very common powerful text editor
emacs	another very common powerful text editor
kate	simple editor with a graphical interface
nano	simple editor run in a shell and with context help

Since most files we will work with are simple text files, text editor is a necessary tool to view and modify them. We list some common text editors in Table 4 Most widespread and powerful text editors under Linux are `vi` and `emacs`. We recommend that you learn to use one of them. They may not be intuitive at the beginning but when you learn to use them, they save you a lot of time and trouble. Just type the editor name in a command line and try it out. *Note, that for example M\$ Word or OpenOffice.org Writer are not text editors but word processors, which is something completely different! Just look at Text Editors page of Wikipedia to find out more.*

3.4 Plotting: GNUPLOT and xmgrace

Data from simulations and also data in Science in general are often presented in a form of plots. Therefore, one needs an appropriate plotting program. One that we recommend is GNUPLOT. It is relatively easy to use and offers a broad functionality particularly suited for scientific applications. In principle, also powerful mathematical programs such as Maple or Matlab can be used for plotting and processing data, however, they are primarily designed for symbolic math computations and also not so easy to learn. A good plotting program with a graphical frontend is `xmgrace`.

In principle, one could also use M\$ Excel or OpenOffice.org Calc to plot the data. But these programs are not primarily designed for plotting scientific data and offer very limited possibilities in this respect (besides being utterly slow).

3.5 Suggested tasks

- Using one of the text editors, create a text file which contains fake data (numbers) in three columns and five rows.
- Plot the data with GNUPLOT
- Use `ssh` to connect to a neighbouring computer of the CIP pool.
- If you have an M\$ Windows PC, download Putty and try connecting to `gate.cip.ica.uni-stuttgart.de` using `ssh`. You may also download and install GNUPLOT, `vi` and `emacs`.
- If you do not have Linux on your home computer, you may try installing it.

4 Programming in C

Among a number of programming languages, C programming language is a common choice for writing simulation programs. All tutorial source codes are written in C.

4.1 Basic syntax

Describing syntax of any programming language is a topic for a book. In the following, we give you a few guidelines which could help you read and understand code in C, especially if you have been using some other programming language. Learning to write programs in C is beyond the scope of the course.

Variables

Variables in C have to be declared before use. The declaration has to define a variable type, its name and optionally its initial value. For example

```
int i, Index_of_element3=14;
double MyPi=3.14;
```

I declare two integers named `i` and `Index_of_element3` and a double named `MyPi` (i. e. a number with a floating decimal point with a double precision) with an initial value of 3.14. Note that after each statement, a semicolon, ";" has to be inserted as a marker of its end. To assign a value to a variable, we use

```
i=10;
energy=10.003;
energy=1.27e-3
```

In C, one can perform basic mathematical operations (+,-,*,/,%) with variables, such as

```
i=10+j;
energy=energy*2.7;
```

where the result of operation of the right of the "=" sign is assigned as a new value to the variable on the left. As a shorthand notation, following operations are equivalent

```
i=10/i;
i/=10;
```

To produce the expected output, variables have to be of a proper type. There are well rules for what happens different variable types are combined in one operation, but this is beyond the scope of our tutorial.

Comments can be inserted into C code like that: `/* comment */`.

```
i=10/3; /* i now has the value 3*/
energy=10.0/3; /* energy now has the value 3.33333...*/
i = 10%3; /* i now has the value 1 (remainder of the integer division 10/3) */
```

Arrays

An array is essentially a matrix, containing variables of a given type, e. g.

```
double myArray[5] = { 1.2, 2.1, 3.7, 4.3, 5.0};
```

declares a variable `myArray` with five elements of type `double`, and at the same time assigns to them values in curly brackets. Members of an array can be accessed by indices and manipulated in the same way as other variables. Indexing of an array in C begins always with zero. For example,

```
d=myArray[0];
```

assigns to `d` value of the first element in `myArray`, which is 1.2.

Flow control: Loops and if/else

Loops are widely used in all simulation programs to execute some commands repeatedly. For example

```
for(i=0;i<10;i++) {
    printf("i: %d, 10*i: %d\n",i,i*10);
}
```

prints value of `i` and `10*i` on one line for all values of $i = 0, \dots, 9$.

C commands and man pages

Note that you can find description of all standard C calls within the man-pages. Be careful to select the proper section of the manuals, as `man 1 printf` calls the man-page for the shell command `printf`, whereas `man 3 printf` calls the man-page for the C function with the same name.

Functions

Another important concept in C is a function. A function usually takes some arguments, performs some operations and returns a value. For example

```
int max(int i, int j) {
    if (i>j) return i;
    else return j;
}
```

is a function which takes two arguments of type `int` and returns the one, which has a bigger value.

First program: hello world

Some often used functions in C are pre-defined and can be found in libraries. An example is function `printf` which is in the `stdio` standard C library and is used to print formatted text to standard output (the declaration of the function is in the `stdio.h` header file, and has to be included using what is called a pre-processor directive!). We will use it to produce a simple program:

```
#include <stdio.h>
int main() {
    printf("Hello world\n");
    return 0;
}
```

Each C program contains a function called `main` which is called when the program is executed. In our case, the program does one single thing: it prints the text

```
Hello world
```

to the standard output.

4.2 Compilation

To obtain an executable file out of a source code, one has to compile it, i. e. to convert the text of the source to instructions for the hardware. This job is done by a program called compiler. To compile a C source on a Linux, one would usually use e. g. `cc` or `gcc`. For example, to create an executable file `myprogram` from sources `source1.c`, `source2.c`, `source3.c`, one would use

```
cc source1.c source2.c source3.c -o myprogram
```

After changes to the sources have been made, one always needs to re-compile a program to make the changes effective.

4.3 Make and Makefile

In more complex programs, one can compile several source files into an object file, which can be used later on for compilation. Typically, a larger program is split into many source files and many object files. When one of the source files is changed, one only needs to recompile those files, which directly depend on it. For this purpose, program `make` comes in very handy. It searches for a file named `Makefile` in which rules for dependencies between various source files are defined. According to these rules it recognizes which parts of the code need to be re-compiled in order to get an up-to-date version of the executable.

In most tutorials, you are presented with a program written in C, in which you are supposed to implement several lines of code, e. g. integration or force calculation routines. Other parts of the program, such as reading in the input data, computing observables and writing output to files have been already implemented by someone else. The code is provided together with a `Makefile` which enables you to compile the program just by typing

```
make
```

without the need to know details about those parts of the program which do not concern your task.

4.4 Suggested tasks

- Write, compile and run the “Hello world” program.
- Write a simple program which adds numbers.
- Write a program which computes the value of an $e^{-3.273561}$ by summing up its Taylor series. Look at how it converges to the true value with increasing order of truncation.