Unix Tutorial 1

Dr. Yihsiang Liow   (September 9, 2022)

Contents

1 Introduction

What is an operating system?

"An operating system (OS) is a set of computer programs that manage the hardware and software resources of a computer. An operating system rationally processes electronic devices in response to approved commands. At the foundation of all system software, an operating system performs basic tasks such as controlling and allocating memory, prioritizing system requests, controlling input and output devices, facilitating networking, and managing files. Most operating systems have a command line interpreter as a basic user interface, but they may also provide a graphical user interface (GUI) for higher level functions. The operating system forms a platform for other system software and for application software."

- from http://www.wikipedia.org

What is Unix?

"Unix (officially trademarked as UNIX®) is a computer operating system originally developed in the 1960s and 1970s by a group of AT&T employees at Bell Labs including Ken Thompson, Dennis Ritchie and Douglas McIlroy. Today's Unix systems are split into various branches, developed over time by AT&T as well as various commercial vendors and non-profit organizations."

- from http://www.wikipedia.org

What is Linux?

"Linux is a Unix-like computer operating system assembled under the model of free and open source software development and distribution. The defining component of Linux is the Linux kernel, an operating system kernel first released October 5, 1991 by Linus Torvalds.

"Linux was originally developed as a free operating system for Intel x86-based personal computers. It has since been ported to more computer hardware platforms than any other operating system. It is a leading operating system on servers and other big iron systems such as mainframe computers and supercomputers:more than 90% of today's top 500 supercomputers run some variant of Linux, including the 10 fastest. Linux also runs on embedded systems (devices where the operating system is typically built into the firmware and highly tailored to the system) such as mobile phones, tablet computers, network routers, televisions and video game consoles; the Android system in wide use on mobile devices is built on the Linux kernel."

- from http://www.wikipedia.org

For this tutorial, I assume you have access to a Unix/Linux system. So at least one of the following is true:

- You have the Fedora virtual machine that I built to run in VMware player/workstation. I assume you have already read the document *VMware Player/Workstation and Fedora Virtual Machine*.

- You have a user account at `gandalf.ccis.edu`. You have read my *PuTTY tutorial* and know how to use PuTTY (or some other SSH client) to login to your account at `gandalf.ccis.edu`. (Or maybe you have a Unix/Linux account at another server.)

- You have access to Cygwin on a Windows machine. (Cygwin is not really Unix but a Unix emulation.) I assume you have read my *Cygwin Tutorial*.

- You have access to your own linux machine.

I'll be using a Linux system. However everything you learn in this tutorial should apply to any Unix/Linux system.

Also, I will be interacting with the Linux machine through a terminal shell. So make sure you have a terminal window open right away.
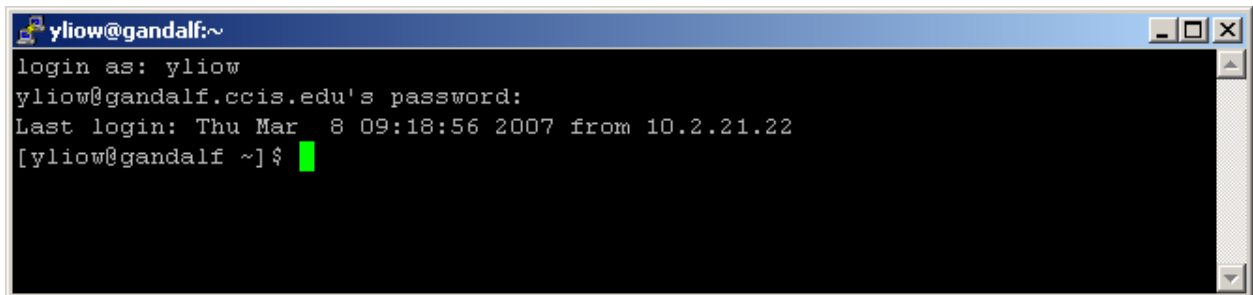
For the first option, i.e., using the Fedora virtual machine I built, the following shows the terminal shell after I double-click on the terminal icon on the Desktop in the virtual machine:
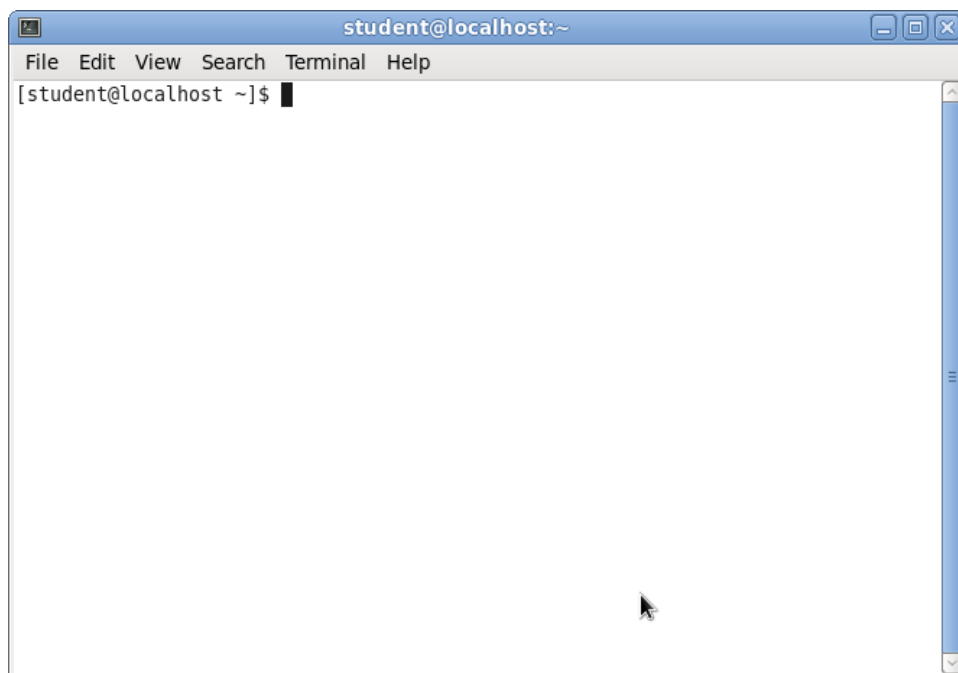
You can see from the above that the **prompt** is

```
[student@localhost ~]
```

For the second option, i.e., using PuTTY to login to an account on a remote linux server, the following is the window I see after I login to `gandalf.ccis.edu`:

The prompt is

```
[yliow@gandalf ~]
```

For the third option, i.e., using Cygwin to emulate a linux environment, the following is the window I see after running Cygwin

The prompt is

```
$
```

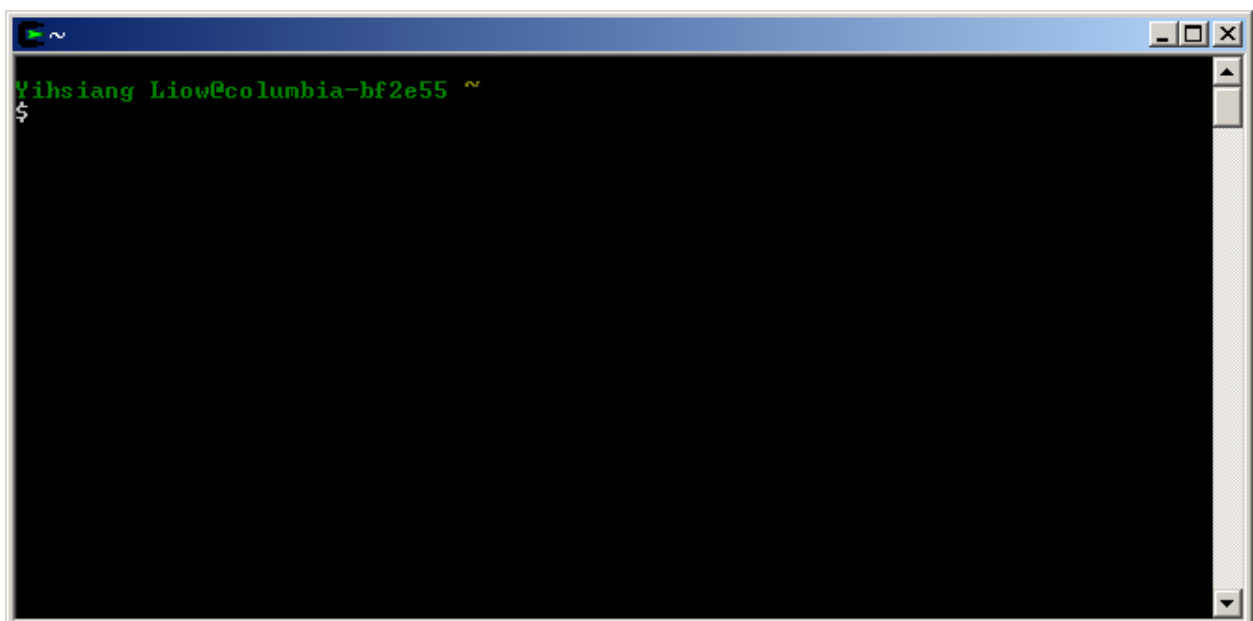There are too many possible variations for the fourth option. If you're using the fourth option, you're entirely on your own.

For this tutorial, I will not use screen shots. Instead of a screen shot, I will say something like: "Enter the following command":

```
$ ls
```

This means you enter the command `ls` at the shell prompt and press the `[enter]` key. I will be using $ to denote the prompt. To log out from the shell, you just do:

```
$ exit
```

Note that we will not be using the mouse at all. Although I will be showing you how to interact with Linux through the command-line interface via a console window, most Linux distributions do come with GUI (icons, windows, mouse, etc.) With practice, you will find that the shell is a lot more productive than clicking around with the mouse. In due time, you will find that you won't be using the mouse that much. So stop whining.

Linux is a complex operating system. The purpose of this tutorial is to teach you just enough to write and run C/C++ programs.

OK. Let's begin.

2 Changing Your Password

This is how you change your password:

```
$ passwd
```

It is your responsibility to ensure that you have a well chosen password.

A very good way of selecting a password is to select a phrase that is easy to remember and then picking characters from that phrase. For instance I might select the phrase:

```
the answer to life, universe, and everything is forty-two
```

I then select the first character of each word to create this password:

```
tatluaeift
```

You can change some of the above to numeric characters:

```
tatluaei4t
```

You can also include punctuations:

```
tatl,u,aei4t
```

You can capitalize some of the characters:

```
tatL,U,aEi4t
```

(No, that's not one of my passwords).

Here are some suggestions for choosing good passwords: At least 24 characters (no I'm not paranoid, I'm just not naïvely optimistic) and no recognizable substrings such as words, dates, acronyms, etc. The following are bad passwords although they are long:

```
mynameisjohndoeandihaveaverylongpassword
johndoe12251990janesmith04011988
youcantbreakmypasswordlol
```

You are not allowed to modify your given password for your user account at `gandalf.ccis.edu`.

Exercise. Choose a very good password, change your password, logout, login with the new password to verify that it's changed. [If this is your account at `gandalf.ccis.edu`, you should change the password back to what it was. Verify that you did change it to the original password by logging out and logging back in with the original password.] □

Summary

`passwd`	`change password`

3 Basic File Directory Commands

Just like Windows XP or Windows 7 where there are folders and regular files, the Unix OS has directories and regular files. A Unix **directory** is similar to a Windows folder. When you logged in to your unix machine (either on `gandalf.ccis.edu`, your virtual machine, etc), you are placed in a certain directory. This is your **home directory**.

Do this to see the complete name of the directory you are currently in:

```
$ pwd
```

This is the "print working directory" command. For me the output is

```
/home/yliow
```

This means the following:

```
    /            is the root directory
    /home        is the home directory in the root directory
    /home/yliow  is the yliow directory in the home directory in the root
                 directory.
```

Note that the first / is the **root directory** while the second / (the one between `home` and `yliow`) acts as a separator of `home` and `yliow` and is not a directory.

```
    /home/yliow
```

is my **home directory** because when I login to `gandalf.ccis.edu` by default I am placed in this directory. This picture might help:

```
/
|
+-- home
|     |
|     +-- yliow        <--- I AM CURRENTLY HERE
|     |
|     |
.     .
.     .
.     .
```

Of course there are other directories including other user home directories under the `/home` directory. If your login name is `jdoe`, then your home directory is `/home/jdoe`:

```
    /
    |
    +-- home
    |     |
    .     .
    .     .
    .     .
    |     +-- jdoe        <--- John Doe's home directory
    .     .
    .     .
    .     .
    |     +-- yliow
    |     |
    |     |
    .     .
    .     .
    .     .
```

Now do

```
$ ls
```

to view the contents of your home directory. This is the **list directory contents** command. (Sometimes I will say "list" or "list directory".)

If this is the first time you're accessing your Unix account you won't see anything since you haven't created any files. However there are some hidden files which are automatically created when your user account was created. To show the hidden files and also to get all the information from each file, you do this:

```
$ ls -la
```

and you will see something that looks like this:

```
    drwxr-xr-x   9 yliow yliow 4096 Mar  7 21:53 .
    drwxr-xr-x  38 root  root  4096 Feb 13 18:01 ..
    -rw-------   1 yliow yliow  847 Mar 10 10:12 .bash_history
    -rw-r--r--   1 yliow root    24 Mar  6 10:29 .bash_logout
    -rw-r--r--   1 yliow root   191 Mar  6 10:29 .bash_profile
    -rw-r--r--   1 yliow root   176 Mar  6 10:29 .bashrc
    -rw-r--r--   1 yliow yliow  328 Mar  5 15:28 .emacs
    drwxrwxr-x   3 yliow yliow 4096 Mar  6 11:19 .emacs.d
```

I will explain the directory listing later. Do this

```
$ ls -la .b*
```

and you will list items starting with `.b`.

If you want to list items ending with `rc` you do this:

```
$ ls -la *rc
```

If you want to list items starting with .b and ending with rc you do this:

```
$ ls -la .b*rc
```

The expressions

```
    .b*            *rc    .b*rc
```

are examples of **regular expressions**. We will not go into this very powerful concept. (Take the Automata theory, CISS362, for more information.)

Exercise 3.1. You don't have any C++ files yet. But suppose you're working on a project and you want to list all the file ending in `.cpp`. How would you do that? □

Now type this

```
$ mkdir tmp
```

This creates a directory called `tmp` from your current directory using the "make directory" command. Now do

```
$ ls -la
```

again and you should see `tmp`:

```
    /
    |
    +-- home
    |     |
    |     +-- yliow        <--- I AM CURRENTLY HERE
    |     |     |
    |     |     +-- tmp
    |     |
    .     .
    .     .
    .     .
```

(I'm not showing the hidden files.)

Let's get rid of this directory. Execute this:

```
$ rmdir tmp
```

This is the "remove directory" command. List your home directory to verify that `tmp` is removed.

Exercise 3.2. Go ahead and create a directory called `cpp` from your home directory. Show the contents of your home directory to verify that you do have a new directory called `cpp`. The structure of the file system from your home directory now looks like this:

```
    /
    |
    +-- home
    |     |
    |     +-- [your home]      <--- YOU ARE CURRENTLY HERE
    |     |     |
    |     |      +-- cpp
    |     |
    .     .
    .     .
    .     .
```

□

Exercise 3.3. Create a directory for all your CISS classes required by your instructor. (Check with your instructor). For instance you might want to create `ciss240` and `ciss245`.

```
    /
    |
    +-- home
    |     |
    |     +-- [your home]
    |     |     |
    |     |      +-- ciss240
    |     |     |
    |     |      +-- ciss245
    |     |     |
    |     |      +-- cpp
    |     |
    .     .
    .     .
```

.   .

□

4 Moving Around the File System

Now let's go into the `cpp` directory with this command:

```
$ cd cpp
```

`cd` is the "change directory" command. You are now in the `cpp` directory of your home directory:

```
/
|
+-- home
|      |
|      +-- [your home]
|      |    |
|      |    +-- ciss240
|      |    |
|      |    +-- ciss245
|      |    |
|      |    +-- cpp        <--- YOU ARE CURRENTLY HERE
|      |
.      .
.      .
.      .
```

You also notice that your prompt has changed. Your prompt shows you your working directory.

Exercise 4.1. Do a print working directory to verify that you are indeed in the `cpp` directory in your home directory. (Don't remember how to do a print working directory? Check previous section.) □

Next do

```
$ ls -la
```

Notice that there is the dot (i.e., .) and the double-dot (i.e., ..). The double-dot refers to the parent directory of your current working directory (i.e. it's one level up). The parent directory of `cpp` is your home directory.

```
    /
    |
    +-- home
    |     |
    |     +-- [your home]     <--- The .. directory of cpp refers to this
    |     |    |
    |     |    +-- ciss240
    |     |    |
    |     |    +-- ciss245
    |     |    |
    |     |    +-- cpp        <--- YOU ARE CURRENTLY HERE
    |     |
    .     .
    .     .
    .     .
```

Execute this
```
$ cd ..
```
and do a print working directory. Are you in your home directory? The answer should be Yes. If not, slap yourself, wake up, pay attention, go to the beginning of this tutorial, and redo everything.

Exercise 4.2. Go back to directory `cpp` again. □

You should be in your `cpp` directory again. We just talked about the double-dot. Now what about the dot? The dot refers to your current directory. So the dot in my `cpp` directory is the same as `/home/yliow/cpp`. So if you do this:
```
$ cd .
```
you don't end up somewhere else! You're still in `cpp`. You might say "What's the point of that?!?" We'll come to that in a bit.

Exercise 4.3. Go to your home directory from `cpp`. From there do a change directory to the dot. Print your working directory that you are indeed still in your home directory. □

Exercise 4.4. As an exercise, create a directory called `secretdiary` in your home directory and another directory called `backup` in your `cpp` directory and go into the `backup` directory. The directory structure from your home directory now looks like this:

```
[your home]            <--- parent of cpp and secretdiary
  |
  |
 +-- ciss240
  |
 +-- ciss245
  |
 +-- cpp               <--- parent of backup
  |    |
  |    +-- backup      <--- YOU ARE CURRENTLY HERE
  |
 +-- secretdiary
```

□

Now let's go to your home directory from `backup`. You need to go to the parent directory of `backup` (i.e. `cpp`) and then go to the parent of that directory. So one way to achieve that is to execute `cd ..` twice (don't do it yet!!!):

```
$ cd ..
$ cd ..
```

Instead of doing that do this right now:

```
$ cd ../..
```

Remember that the `/` is a separator of directories. Print your working directory to verify that you are in your home directory. In general you can enter any path at the `cd` command. For instance if you are in directory `Y`:

```
W
|
+-- X
|   |
|   +-- Y     <--- YOU ARE CURRENTLY HERE
|
+-- Z
```

you can get to `Z` by doing this:

```
$ cd ../../Z
```

and you then have this:

```
W
|
+-- X
|   |
|   +-- Y
|
+-- Z            <--- YOU ARE CURRENTLY HERE
```

Correct?

Exercise 4.5. You should now be in your home directory:

```
[your home]              <--- YOU ARE CURRENTLY HERE
  |
  |
 +-- ciss240
  |
 +-- ciss245
  |
 +-- cpp
  |   |
  |   +-- backup
  |
 +-- secretdiary
```

In one command, go to the `secretdiary`. From there, using one command, go to the `backup` directory (in `cpp`). From there, using one command, go to `ciss240`. Finally, from `ciss240`, using one command, go to the `backup` directory (in `cpp`). □

Let me show you a really handy shortcut for your home directory. First go to the `backup` directory in `cpp` so that:

```
[your home]
  |
 +-- ciss240
  |
 +-- ciss245
  |
 +-- cpp
  |    |
  |   +-- backup     <--- YOU ARE CURRENTLY EHRE
  |
 +-- secretdiary
```

To go to your home directory you can do this (don't do it yet!!!):

```
$ cd ../..
```

There's actually a shorthand notation for your home directory. Do this:

```
$ cd ~
```

That's a tilde. Check that you are in your home directory. The ~ is a shortcut for your home directory.

Exercise 4.6. Go to your `backup` directory in `cpp`. Next go to the `secretdiary` directory in one command without using any `..` notation. □

Summary

```
passwd          change password

pwd             print working directory
cd              change directory
mkdir           make directory
rmdir           remove (nonempty) directory


.               current directory
..              parent directory
~               home directory


ls              list directory
ls -la          list all (include hidden files) in long format
```

5 Path Completion

Let me show you a really handy feature of the shell. First go to your home directory (using ~).

Here's where you are at this point:

```
[your home]              <--- YOU ARE CURRENTLY HERE
  |
  +-- ciss240
  |
  +-- ciss245
  |
  +-- cpp
  |    |
  |    +-- backup
  |
  +-- secretdiary
```

Now I want you to go to `backup` again. But WAIT!!! ... do it like this: First type (don't press the [enter] key yet!!!):

```
$ cd cp
```

and now press the [tab] key. You see that automatically your directory is completed for you:

```
$ cd cpp/
```

now type `b` to get

```
$ cd cpp/b
```

and do [tab] again and you get

```
$ cd cpp/backup/
```

Now press the [enter] key and you're in the `backup` directory. The [tab] key will perform path completion for you. Your shell will only complete words so that it forms a valid path. If there are several possibilities and you press [tab] twice, your shell will list all the possibilities for you. Let's try that.

Go to you `cpp` directory (use `..`) and create directory `bagpipes` in `cpp`.

```
[your home]
   |
 +-- ciss240
   |
 +-- ciss245
   |
 +-- cpp              <--- YOU ARE CURRENTLY HERE
 |   |
 |   +-- backup
 |   |
 |   +-- bagpipes
 |
 +-- secretdiary
```

Go to your home directory. Try this

```
$ cd c
```

press the [tab] to get

```
$ cd cpp/
```

Press `b` to get this

```
$ cd cpp/b
```

Now if you press [tab] the shell will not complete the path for you. If you press [tab] a second time, the shell will show you two options matching `cpp/b`. Press `a` to get

```
$ cd cpp/ba
```

and press [tab]. Again the shell does not complete the path. Press [tab] again and the shell gives you two options. *Now* if you enter `g` to get

```
$ cd cpp/bag
```

and press [tab] the shell will indeed complete the path for you:

```
$ cd cpp/bagpipes
```

Get it?

From now on, *use path completion as much as you can*. Don't be a noob.

Exercise 5.1. Go to the `secretarydiary` directory from the `bagpipes` directory using one command and path completion wherever possible.

Exercise 5.2. Remove the `bagpipes` directory and the `secretdiary` directory. (Remember how to remove directories?) Use path completion!

Summary

```
passwd          change password

pwd             print working directory
cd              change directory
mkdir           make directory
rmdir           remove (nonempty) directory


.               current directory
..              parent directory
~               home directory


ls              list directory
ls -la          list all (include hidden files) in long format


[tab]           path completion
```

6 Emacs and XEmacs

Let's write a C++ program, say a simple hello world program. First go to your `cpp` directory.

Of course you need something to enter your program! Do this:

```
$ emacs helloworld.cpp
```

or

```
$ emacs helloworld.cpp
```

(It depends on whether the Emacs or XEmacs software is available on the Unix system you're using.) If you're using one of our Fedora virtual machine, then Emacs or XEmacs should be installed.

Actually, although there are differences, Emacs and XEmacs are very similar. From now on I'll just say Emacs.

Emacs/XEmacs is a text editor (and more); You can enter text using Emacs/XEmacs just like your Notepad in Windows XP/Vista/10/.... For this document, I will only show you the most basic Emacs/XEmacs commands; refer to the Emacs/XEmacs tutorial for more information. Enter this hello world program in your Emacs/XEmacs window:

```cpp
#include <iostream>

int main()
{
    std::cout << "hello world" << std::endl;
    return 0;
}
```

Now you need to save and exit Emacs/XEmacs. Do this:

```
C-x-s
```

This means that you do the the following in sequence:

```
hold the [ctrl] key down (don't release it yet!)
press and release the x key
press and release the s key.
```

You may then release the [ctrl] key. (Yeah I know ... it's bizarre at first). This will save your program.

Now do

```
C-x-c
```

This will exit XEmacs. This means

```
hold the [ctrl] key down (don't release it yet!)
press and release the x key
press and release the c key.
```

You're back to your command-line interface with the Unix operating system. By the way if you want to save and exit together you need not press-and-release the [ctrl] key twice, i.e. you can do this

```
C-x-s-x-c
```

i.e.,

```
hold the [ctrl] key down (don't release it yet!)
press and release the x key
press and release the s key
press and release the x key
press and release the c key
release the [ctrl] key
```

Now list the contents of your directory and verify that you have `helloworld.cpp`. (How do you do that again?)

You can view the contents of your file in the shell by doing this (using path completion for `helloworld.cpp`):

```
$ less helloworld.cpp
```

`less` is useful when you want to view a file without modifying it. Of course you can open it using Emacs/XEmacs but `less` loads a lot faster. Note that if you have a really long file, `less` will display one screen full at a time. You can scroll up or down using the arrow keys and page up/down key. To exit `less`, you press the `q` key (for quit).

Exercise 6.1. Repeat the above by create `helloworld1.cpp`, `helloworld2.cpp`, etc. until you feel comfortable with using XEmacs. □

Exercise 6.2. There's a hidden file in your home directory called `.bashrc`. (The name of the file starts with a dot.) View the contents of the file using `less`. Scroll up and down using the arrow keys and page up/down keys. Quit `less`. □

Refer to the Emacs/XEmacs tutorial for more information on this software.

Exercise 6.3. Create a file, `unix.txt`, in your home directory containing a summary of all Unix commands; see below. Make sure you keep this file carefully! Keep adding useful information to it! □

Summary

```
passwd          change password

pwd             print working directory
cd              change directory
mkdir           make directory
rmdir           remove (nonempty) directory


.               current directory
..              parent directory
~               home directory


ls              list directory
ls -la          list all (include hidden files) in long format


less            view file content


[tab]           path completion


emacs/xemacs
C-x-s           save
C-x-c           exit
```

7 Compiling and Running C++ Programs

Now we're ready to compile your `helloworld.cpp` to get an executable program. We will use the GNU C++ compiler. Do this (using path completion for `helloworld.cpp`!):

```
$ g++ helloworld.cpp -o hw
```

This creates an executable called `hw` that you can execute. Do an `ls` to check that you have `hw`.

The `-o` in the above command is called the output option.

To run your `hw` program you do this:

```
$ ./hw
```

(Remember `.` means the current directory? This is one of the reasons why there is the dot.)

If you are in a C++ project directory where you want to compile all the C++ files, you compile with

```
$ g++ *.cpp -o hw
```

In this case, the directory should only contain the files for one program.

This is a short introduction to the GNU C++ compiler. Refer to my gcc-g++ tutorial for more information.

Exercise 7.1. Build another executable from your program but name the executable `abc`. Run it. □

Exercise 7.2. Must you run `hw` from the directory containing `hw`? Go to your home directory and run your `hw` executable from your home directory. □

Note that if you want to compile a C program (not C++), say `spam.c` you can do this

```
$ g++ spam.c -o spam
```

or this:

```
$ gcc spam.c -o spam
```


Summary

```
passwd          change password


pwd             print working directory
cd              change directory
mkdir           make directory
rmdir           remove (nonempty) directory


.               current directory
..              parent directory
~               home directory


ls              list directory
ls -la          list all (include hidden files) in long format


[tab]           path completion


less            view file content


emacs/xemacs
C-x-s           save
C-x-c           exit


gcc/g++ (GNU C/C++ compiler)
gcc x.c -o x    Compile C program x.c to executable x
g++ x.cpp -o x  Compile C++ program x.cpp to executable x
```

8 History

Here's another handy feature of the shell.

Sometimes you need to re-execute a previous command. This is the case when you are debugging and recompiling a program. Your shell actually remembers a history of previous commands. To scroll through the history of previous commands, you use the arrow keys. Let's try it.

Do this:
```
$ ls -la
```
Now use the uparrow key and redo the list directory command. Easy right? Remember that you can scroll up and down through your history using the uparrow and downarrow keys.

Exercise 8.1. Go to your cpp. Now start XEmacs with `~/cpp/helloworld.cpp` (using path completion). Insert some errors into your program like so:
```
#include <iostream>

int main
{
    std::cout << "hello world << std::endl;
    return 0
}
```
Save your file and exit XEmacs. Compile your program. Of course g++ yells at you. Using the history of commands, continually alternate between correcting one error at a time with XEmacs and recompiling your code until all errors are resolved.

Summary

```
passwd          change password

pwd             print working directory
cd              change directory
mkdir           make directory
rmdir           remove (nonempty) directory


.               current directory
..              parent directory
~               home directory


ls              list directory
ls -la          list all (include hidden files) in long format


less            view file content


[tab]           path completion


[up/down]arrow  scroll through history


emacs/xemacs
C-x-s           save
C-x-c           exit


gcc/g++ (C/C++ compiler)
gcc x.c -o x    Compile C program x.c to executable x
g++ x.cpp -o x  Compile C++ program x.cpp to executable x
```

9 Man Pages

At this point it's appropriate to teach you to teach yourself. If you want to know more about g++ you can type this at your Unix prompt:

```
$ man g++
```

This will show you the documentation from the g++ man pages (man = manual). You can use the up and down arrow keys and the page up and down (or spacebar) keys to scroll through the documentation. Recall that we executed

```
$ g++ helloworld.cpp -o hw
```

The `-o` is called an option. Look for the section in the documentation describing this option.

To quit man pages, just type `q`.

Exercise 9.1. Search the man pages for `cd`. □

The amount of information can be overwhelming especially for a beginner. But anyway you should be aware of this resource.

Another useful resource is of course ... the web. And there's no better web search engine than google. For a beginner, some keywords you want to include in your web search are "newbie", "tutorial", "howto", "beginner", etc.

Exercise 9.2. Google for "beginner g++ tutorial". □

Summary

```
passwd          change password


pwd             print working directory
cd              change directory
mkdir           make directory
rmdir           remove (nonempty) directory


.               current directory
..              parent directory
~               home directory


ls              list directory
ls -la          list all (include hidden files) in long format


less            view file content


[tab]           path completion


[up/down]arrow  scroll through history


man             man pages


emacs/xemacs
C-x-s           save
C-x-c           exit


gcc/g++ (C/C++ compiler)
gcc x.c -o x    Compile C program x.c to executable x
g++ x.cpp -o x  Compile C++ program x.cpp to executable x
```

10 Directory Information

Here's your part of the file system:

```
[your home]
  |
 +-- ciss240
  |
 +-- ciss245
  |
 +-- cpp                      <--- YOU ARE CURRENTLY HERE
     |
     +-- backup
     |
     +-- helloworld.cpp
     |
     +-- hw
```

Let's talk about directory listings again. Go to your home directory. Using path completion, list the cpp directory. You should get something similar to this:

```
drwxrwxr-x  2 yliow yliow 4096 May 27 03:21 .
drwxr-xr-x  5 yliow yliow 4096 May 27 03:20 ..
-rw-rw-r--  1 yliow yliow   95 May 27 03:21 helloworld.cpp
-rwxrwxr-x  1 yliow yliow 6403 May 27 03:21 hw
```

Each line describes a file (a directory is a file). Some of the data is pretty obvious. For instance from the line

```
-rw-rw-r--  1 yliow yliow   95 May 27 03:21 helloworld.cpp
```

you can easily guess

```
name of file:                  helloworld.cpp
owner of the file:             yliow
datetime of last modification: May 27 03:21
file size:                     95 (bytes)
```

I'll give you all the details in a later Linux tutorial. For this tutorial, I will only mention a few more things.

Let's look at the first column. You see, for instance, something like

```
drwxr-xr-x
```

or

```
-rw-r--r--
```

This is really made up of four parts. Think of, for instance,

```
drwxr-xr-x
```

as being made up of

```
d    rwx    r-x    r-x
```

I will only talk about the first two parts.

```
d    rwx    r-x    r-x
```

The first part is either a "d" or a "-" (there are other cases that I will not talk about here.) When it's a "d" it means that the entry is a directory. This is the case for the dot and the double-dot:

```
drwxrwxr-x  2 yliow yliow 4096 May 27 03:21 .
drwxr-xr-x  5 yliow yliow 4096 May 27 03:20 ..
```

You see that all other regular files have a "-". For instance

```
-rw-rw-r--  1 yliow yliow   95 May 27 03:21 helloworld.cpp
-rwxrwxr-x  1 yliow yliow 6403 May 27 03:21 hw
```

Not too bad right?

Now look at the next three characters of the first column:

```
-rwxrwxr-x  1 yliow yliow 6403 May 27 03:21 hw
```

i.e.,

```
rwx
```

It means that you can "read", "write" and "execute" this file. For the file

```
-rw-rw-r--  1 yliow yliow   95 May 27 03:21 helloworld.cpp
```

You see that you can "read" and "write" to it; but you cannot execute the `helloworld.cpp` file.

That's all you need to know right now.

11 Changing File Permission

The first chunk of

```
rwx
```

from the previous section is called the user file permission. You can actually change them. And that's what I will talk about in this section.

First verify that you can read the contents of the `helloworld.cpp` by doing this:

```
$ less helloworld.cpp
```

No problems, right? You can also verify that you have read access to your `helloworld.cpp` doing:

```
$ ls -l helloworld.cpp
```

You should see this something like this:

```
-rw--rw-r--  1 yliow yliow   95 May 27 03:21 helloworld.cpp
```

OK. Now I want you to execute this:

```
$ chmod u-r helloworld.cpp
```

The `chmod` is the "change mode" command. You pronounce it as "shh-mod". In the above command you see `u-r`. This is what it means:

```
u = user    - = remove    r = read
```

In other words "remove read permission from user".

Now when you do

```
$ ls -l helloworld.cpp
```

you will see

```
--w--rw-r--  1 yliow yliow   95 May 27 03:21 helloworld.cpp
```

Notice that the first `r` has disappeared. And when you attempt to read the file:

```
$ less helloworld.cpp
```

you see this

```
helloworld.cpp: Permission denied
```

In short: You do not have read access to `helloworld.cpp`.

Now I want you do execute this:

```
$ chmod u+r helloworld.cpp
```

Here's the meaning of `u+r`:

```
u = user    + = add    r = read
```

In other words "add read permission to user". Verify using `ls -l helloworld.cpp` that you have read access to the file and then do a `less` on the file.

What about write access? Do this:

```
$ chmod u-w helloworld.cpp
```

This will remove your write access to `helloworld.cpp` (duh.)

To verify that you cannot write to the file, open `helloworld.cpp` with XEmacs and try to modify the contents.

Next, exit XEmacs and execute:

```
$ chmod u+w helloworld.cpp
```

If you can't guess what the above does, go ahead and redo the whole tutorial after a cup of coffee.

Exercise 11.1. Verify that you have write access to `helloworld.cpp`. □

Now why do you need to know all the above?

There are times when your file permission might be incorrect. This can happen when a sneaky program changes the permission behind your back or you download a cpp file and the download software sets the permission incorrectly.

Exercise 11.2. Build an executable from `helloworld.cpp`. Do `ls -la` to look at the permission of the executable. Run your executable. Now change the permission on the executable by removing the `x` (execute) permission. Try to run the executable again. Fix the file permission so that you can run it again. □

Notice the directories also have permissions. I'll go more into this in another tutorial. For now, just make sure that your working directories (where you're writing programs) are all set to allow `rwx`. If for some reason you cannot do something in a directory `xyz`, you should do `ls -l xyz` to check the permission of this directory and, if necessary, execute

```
$ chmod u+rwx xyz
```

In most cases, that would fix the problem.

Summary

```
passwd          change password

pwd             print working directory
cd              change directory
mkdir           make directory
rmdir           remove (nonempty) directory


.               current directory
..              parent directory
~               home directory


ls              list directory
ls -la          list all (include hidden files) in long format


less            view file content


[tab]           path completion


[up/down]arrow  scroll through history


man             man pages


chmod           change file permission


emacs/xemacs
C-x-s           save
C-x-c           exit


gcc/g++ (C/C++ compiler)
gcc x.c -o x    Compile C program x.c to executable x
g++ x.cpp -o x  Compile C++ program x.cpp to executable x
```

12 Changing File Ownership

Besides wrong user file permission, another glitch might occur. You mgiht find that you're not the owner of the file and the permission is set in such a way that only the owner have full access to the file.

What do you do?

Suppose the file in question is `xyz.cpp`. First, do

```
$ ls -l xyz.cpp
```

to verify that the owner is not yourself.

If you have root (or superuser) access to your machine (i.e., you have the root password), you login as root:

```
$ su
```

and enter the root password. Next, as root, you set the owner of `xyz.cpp` to yourself, say `jdoe`, like this:

```
$ chown jdoe xyz.cpp
```

This is the "change owner" command.

Now when you do

```
$ ls -l xyz.cpp
```

you will find that the user who owns the file is `jdoe`.

The last thing to do is to logout of root:

```
$ exit
```

so that you're back to your usual user.

It's important not to stay as root for too long.

Of course the above works assuming you have the root password.

Summary

```
passwd          change password


pwd             print working directory
cd              change directory
mkdir           make directory
rmdir           remove (nonempty) directory


.               current directory
..              parent directory
~               home directory


ls              list directory
ls -la          list all (include hidden files) in long format


less            view file content


[tab]           path completion


[up/down]arrow  scroll through history


man             man pages


chmod           change file permission
chown           change file owner


emacs/xemacs
C-x-s           save
C-x-c           exit


gcc/g++ (C/C++ compiler)
gcc x.c -o x    Compile C program x.c to executable x
g++ x.cpp -o x  Compile C++ program x.cpp to executable x
```

13 More File System Commands

Here's your part of the file system:

```
[your home]
 |
 +-- ciss240
 |
 +-- ciss245
      |
      +-- cpp                    <--- YOU ARE CURRENTLY HERE
          |
          +-- backup
          |
          +-- helloworld.cpp
          |
          +-- hw
```

You already know how to traverse the file directory tree, create and delete directories, and create a text file. In this section I'll show you how to copy, delete, and move files.

Let's copy our C++ program to our `backup` directory. Here's how you do it (use path completion! Twice!):

```
$ cp helloworld.cpp backup/
```

(Obviously `cp` is the copy command. Duh.)

Now do a listing of `backup` (use path completion!) to verify that backup now contains `helloworld.cpp`.

The structure of your file system should look like this:

```
[your home]
 |
 +-- ciss240
 |
 +-- ciss245
 |
 +-- cpp                     <--- YOU ARE CURRENTLY HERE
     |
     +-- backup
     |    |
     |    +-- helloworld.cpp
     |
     +-- helloworld.cpp
     |
     +-- hw
```

Now do this:

```
$ cp helloworld.cpp helloworld2.cpp
```

(Did you use path completion?) List the `cpp` directory. You should now have two cpp files.

So this copy command copies your file to another file.

```
[your home]
 |
 +-- ciss240
 |
 +-- ciss245
 |
 +-- cpp                      <--- YOU ARE CURRENTLY HERE
    |
    +-- backup
    |    |
    |    +-- helloworld.cpp
    |
    +-- helloworld.cpp
    |
    +-- helloworld2.cpp
    |
    +-- hw
```

Now let's backup again. We can use regular expressions to copy one or more files:

```
$ cp h*.cpp backup/
```

(Did you use path completion?) Check that your backup directory now has two cpp files.

```
[your home]
 |
 +-- ciss240
 |
 +-- ciss245
 |
 +-- cpp                      <--- YOU ARE CURRENTLY HERE
    |
    +-- backup
    |    |
    |    +-- helloworld.cpp
    |    |
    |    +-- helloworld2.cpp
    |
    +-- helloworld.cpp
    |
    +-- helloworld2.cpp
    |
    +-- hw
```

Let's remove a file.

Do this

```
$ rm helloworld.cpp
```

(You used path completion right?) Check that the file is deleted by doing a list directory. What if you actually need that file? No problem. Copy from the backup:

```
$ cp backup/helloworld.cpp .
```

(Path completion?) Here's another use of the dot. Don't forget that the dot means "here". Verify you have recovered your `helloworld.cpp` by listing your `cpp` directory.

You are warned that the shell does not (by default) have a recycle bin! So make sure you know what you're doing when you're doing an `rm`.

Exercise 13.1. To copy a directory you use recursive copy `cp -r`. Here's how you do it. Create a directory called `dira` and put some files into the directory. Now execute the command `cp -r dira dirb`. Check that you now have a new directory `dirb` and the contents of `dirb` is exactly the same that of `dira`. □

Instead of copying a file, you might want to move one or more files. Let's move the cpp files from cpp to the backup directory:

```
$ mv *.cpp backup/
```

(Path completion? No? Slap yourself five times.) Verify that there are no cpp files left in your cpp directory. Of course you can move one file at a time but who would do that when you can use regular expressions?

You can also move a whole directory. Let's move the `backup` directory to your home directory:

```
$ mv backup/ ~
```

(You know what I'm going to ask.) At this point you should have this:

```
[your home]
  |
 +-- backup
  |    |
  |    +-- helloworld.cpp
  |    |
  |    +-- helloworld2.cpp
  |
 +-- ciss240
  |
 +-- ciss245
  |
 +-- cpp                     <--- YOU ARE CURRENTLY HERE
      |
      |
      +-- helloworld.cpp
      |
      +-- helloworld2.cpp
      |
      +-- hw
```

Exercise 13.2. What's the difference between

```
$ mv backup/ ~
```

and

```
$ mv backup/* ~
```

Verify your guess by do this:

```
$ mv ~/backup/* ~/ciss240/
```

□

Exercise 13.3. By the way renaming is just moving. Go to your `backup` directory (which is now in your home directory) and rename `helloworld.cpp` to `h.cpp` using `mv`.   □

One more thing. You feel brave: Try to remove the `backup` directory:

```
$ rmdir ~/backup/
```

(Do you need to slap yourself?) It won't work. From the error message you know why. Now try this:

```
$ rm -rf backup
```

or

```
$ rm -r -f backup
```

This is pretty dangerous. It removes `backup` and *everything* in it. Remember that there is no recycle bin in the shell!!! Make sure you really want to remove a whole directory subtree before hitting the [enter] key!!!

Exercise 13.4. Search the man pages for `rm` and check the meaning of options `-r` and `-f`. □

Exercise 13.5. Clean up your files using regular expressions and path completion wherever possible so that you have this:

```
[your home]
 |
 +-- ciss240
 |
 +-- ciss245
 |
 +-- cpp
```

□

Summary

```
passwd          change password

pwd             print working directory
cd              change directory
mkdir           make directory
rmdir           remove (nonempty) directory

cp              copy
cp -r           recursive copy
mv              move
rm              remove
rm -rf          recursive force remove

.               current directory
..              parent directory
~               home directory

ls              list directory
ls -la          list all (include hidden files) in long format

less            view file content

[tab]           path completion

[up/down]arrow  scroll through history

man             man pages

chmod           change file permission
chown           change file owner

emacs/xemacs
C-x-s           save
C-x-c           exit

gcc/g++ (C/C++ compiler)
gcc x.c -o x    Compile C program x.c to executable x
g++ x.cpp -o x  Compile C++ program x.cpp to executable x
```

14 Tar and gzip

One thing that happens frequently is when you need to email some files to someone else. But it's not just the files. Suppose the files are in a directory and in subdirectories of that directory. Suppose you want to send the directory structure as well. Let me show you how to do it.

Go ahead and create a `tmp` directory for this experiment. And in `tmp` create the following directories with some files with random data:

```
tmp
  |
  +-- a01
       |
       +-- a01q01
       |   |
       |   +-- main.cpp
       |
       +-- a01q02
       |   |
       |   +-- main.py
       |
       +-- a02q03
            |
            +-- main.tex
```

Now go back to directory `tmp`, do this:

```
$ tar -cvf a01.tar a01
```

This gives you the file `a01.tar`. Check for yourself that you have this file. This file contains all the information in directory `a01`, including the directories and all the files.

To gzip `a01.tar` (so that it's smaller), do this:

```
$ gzip a01.tar
```

Check that you have the file `a01.tar.gz`. Now you can email `a01.tar.gz`.

By the way, a quick warning. In your directory `a01`, you might have some binary executable programs. Many email servers will reject files that contain binary executables. So you want to remove them before you tar-and-gzip them for emailing.

Now do this experiment. Remove your directory `a01`.

```
$ rm -rf a01
```

Don't forget you still have `a01.tar.gz`. Let's extract all the contents of this file:

```
$ gunzip a01.tar.gz
```

This will give you `a01.tar`. Then do

```
$ tar -xvf a01.tar
```

Voilà ... you have your `a01` again.

15 Input and Output Stream Redirection

You should know by now that developing programs involves testing your code (probably more than once!). It's annoying to enter the same inputs again and again. This is especially the case when the input is long. Here's a very useful feature in Unix.

First go to your `cpp` directory.

Exercise 15.1. Write a program, `sumprod.cpp`, that prompts the user for two integers and print the sum, and then prompts the user for two integers and print the product. The program should print the two integers separated by a space; the program should also print a newline after the second integer. Compile it to get the executable `sumprod`. Test it and make sure it works. □

Now instead of entering inputs when you run the sumprod program do this: First create a file called `input.txt` with XEmacs. In this file enter the following text

```
3 5 2 4
```

Save and exit XEmacs.

Now execute this:
```
$ ./sumprod < input.txt > output.txt
```

Now use `less` to view the contents of `output.txt` (use path ... you know).
```
$ less output.txt
```

Vóila! The program ran without you entering any input! Well, the reason is the

```
    < input.txt
```

sends the data in `input.txt` as input to your program and the

```
    > output.txt
```

sends the output of your program to `output.txt`.

It doesn't matter if you swap the input and output file, i.e., you can also execute
```
$ ./sumprod > output.txt < input.txt
```

Summary

```
passwd          change password


pwd             print working directory
cd              change directory
mkdir           make directory
rmdir           remove (nonempty) directory


cp              copy
cp -r           recursive copy
mv              move
rm              remove
rm -rf          recursive force remove


.               current directory
..              parent directory
~               home directory


ls              list directory
ls -la          list all (include hidden files) in long format


less            view file content
diff            compare files


[tab]           path completion


[up/down]arrow  scroll through history


man             man pages


chmod           change file permission
chown           change file owner


prog <in >out   send input from file in to prog and send output to out


emacs/xemacs
C-x-s           save
C-x-c           exit


gcc/g++ (C/C++ compiler)
gcc x.c -o x    Compile C program x.c to executable x
g++ x.cpp -o x  Compile C++ program x.cpp to executable x
```

16 Comparing Files

A very useful utility is `diff` which allows you to compare files. Try this. Create two files, `a.txt` and `b.txt`:

```
1 2 3
4 5 6
7 8 9
```

```
1 2 3
4 5
7 8 9
```

(Don't insert a blank line at the end of the files.) Save the files and do this:

```
$ diff a.txt b.txt
```

and you should see this output from `diff`:

```
2c2
< 4 5 6
---
> 4 5
```

Can you guess what `diff` is telling you? It is saying:

```
Line 2 of the left file (a.txt) has been changed
to line 2 of the right file (b.txt)
```

Now modify `a.txt` by adding a blank line:

```
1 2 3

4 5 6
7 8 9
```

Do a `diff` again:

```
$ diff a.txt b.txt
```

and you see this:

```
2,3c2
<
< 4 5 6
---
> 4 5
```

Here's how you read the `diff` output:

```
Line 2 and 3 of the left file (a.txt) has been changed
to line 2 of the right file (b.txt)
```

Get it now?

Another useful utility is meld. Go ahead and try

```
$ meld a.txt b.txt
```

Nice right?

Exercise 16.1. Now make `a.txt` and `b.txt` *exactly* the same. Run `diff` on the two files again. What is the output of `diff` for two exact same files? □

Besides the `c` in the output, you might see `a` and `d`. This is what they mean:

```
a - added
d - deleted
```

`diff` can be used to automate program testing. For instance say you have written a function and you want to test it. You can write a program that calls the function and print the return value. Let's say that the test program requires input from the user and you keep the input in input.txt. (You can of course test the function more than once.) Next you compute by hand the expected value and write it to a file, say `correct.txt`. Let's say the executable of the test program is test. You can then execute the following in your shell.

```
$ ./test < input.txt > output.txt
$ diff output.txt correct.txt
```

Exercise 16.2. Use `diff` to test your `sumprod` program from the previous section. The file `correct.txt` looks like this:

```
8 8
```

(Read the requirements carefully: There is a space between the two 8s and there is a newline immediately after the second 8.) ☐

Exercise 16.3. Using the web, find out more about `diff`. ☐

Summary

```
passwd          change password

pwd             print working directory
cd              change directory
mkdir           make directory
rmdir           remove (nonempty) directory

cp              copy
cp -r           recursive copy
mv              move
rm              remove
rm -rf          recursive force remove

.               current directory
..              parent directory
~               home directory

ls              list directory
ls -la          list all (include hidden files) in long format

tar -cvf        create an archive of a directory
tar -xvf        extract an archive
gzip            compress file
gunzip          uncompress file

less            view file content
diff            compare files

[tab]           path completion

[up/down]arrow  scroll through history

man             man pages

chmod           change file permission
chown           change file owner

prog <in >out   send input from file in to prog and send output to out

emacs/xemacs
C-x-s           save
C-x-c           exit

gcc/g++ (C/C++ compiler)
gcc x.c -o x    Compile C program x.c to executable x
g++ x.cpp -o x  Compile C++ program x.cpp to executable x
```

17 What Now?

We have only scratched the surface! Unix is a very powerful operating system.

Knowing Unix/Linux well will make you extremely productive. For instance it's possible to issue one shell command to scan through all your files from your home directory (including subdirectories), making a backup copy of all cpp files so that `x.cpp` is copied to `x.cpp.old` and at the same time renaming all variables `abc` in `x.cpp` to `new_abc`. There are of course many other Unix/Linux commands. Even for the commands in this tutorial, I have not fully gone into all of their full power. Refer to my other tutorials.

Also, most of the Unix/Linux commands have not changed for decades. This means that once you know a Unix/Linux command well, you will know it for life. This is different from GUI – every few years you will have to learn the new GUI.

Unix/linux operating systems are also known to be extremely reliable.

Of course you will need to remember all the commands well so there is a small learning curve. However if you can remember birthdate and telephone/cellphone numbers and email addresses, you can remember linux commands. It's just a matter of practice. With continual use, Unix/Linux will be second nature in no time.

Exercise 17.1. Delete all the directories/files created in this tutorial and redo the whole tutorial. Repeat this for a couple of days. □

There are lots of online references for learning Unix. So make full use of the web if you really want to know Unix/Linux well. There are also tons of books out there on Unix/Linux for different audiences.

Exercise 17.2. Google for "linux beginner tutorial". Save about 5 well written tutorials on your hard drive. Learn one new linux command every week or two. □

There are also lots of very active linux user group on the web. Most of your linux–related questions can be found by googling. If not, you might want to search within a linux web forum. If you can't find an answer to your linux question, post your question to the appropriate message board. Here's my rule: If you have benefitted from a forum, then you should contribute back to the forum. For every one of your question answered, try to answer at least three questions posted to the forum.

Exercise 17.3. Google for "linux newsgroup user group community". Look for one that has a message board for beginners and one that has a large following that respond to

questions quickly. Join the group. □

Depending on the instructor some of our courses also use Unix/Linux. Here are some examples.

In Operating Systems, you will learn more about operating systems, in particular the Unix operating system.

In Computer Networks you learn Unix network/socket programming (among other things).

You might learn to use Unix in courses such as Programming Languages and Compiler Construction since Unix systems come with lots of compilers, interpreters, and compiler construction tools.

One of the most common activity when using a computer system is searching for information or applying an action to selected entities (example: directories, files, processes). Regular expressions are used for such activities. This is a very powerful tool. You will learn more about regular expressions (and more) in Theory of Automata, and probably also in Programming Languages and Compiler Construction.

Regular expressions are closely related to finite state machines/automata. You will learn more about finite state machines/automata in courses such as Automata theory, Programming Languages, Computer Architecture, Artificial Intelligence, etc.