Califf

Linux Information Handout for IT 279

All work for this course is expected to be completed on the School's Linux machines in C++ using the GNU C++ compiler. This handout is for your reference in learning to work in this environment.

Once you are familiar with the Linux commands and operating system, work through the demo exercise under the "Running and Testing Your C++ programs" header. It will benefit you to at least skim the command list before you move on to the demo exercise.

Accessing the Linux Terminal Servers

Access can be done in three different ways. In every case, you must log in using your ULID and your usual password for your university account.

The first two options will give you a full virtual desktop with windows, menus, etc. One option is to use any browser to go to login.it.ilstu.edu and click on the middle button (the one that says Terminal Servers). Then you will need to log in using your ULID and corresponding password. On the resulting page, expand Linux Terminal Servers under ALL CONNECTIONS, and select one of the choices there. You'll then be presented with a desktop in the browser. This option works pretty well and does not require a VPN connection when off campus. However, it is sometimes a little sluggish compared to the second option when on campus.

The second option is to use Remote Desktop on campus. You can simply put in the machine name (such as ash) for connecting to a PC. You will have to accept an issue with authentication. This method requires a VPN connection if you are off campus (see How to Use Cisco AnyConnect VPN Client | IT Help - Illinois State). I recommend sticking with the login.it.ilstu.edu method when off campus, since it tends to be less sluggish than a remote desktop connection through the VPN in my experience.

You can also connect using ssh with a program such as putty, but that will only give you a basic terminal window for command line work.

You may also get a full VS Code window running locally that hosts an SSH connection to the IT Linux Terminals through the Remote-SSH extension. Instructions are in a separate document. Note that this will also require an active VPN connection.

Command line vs. GUI

Except when you choose the ssh option you will have access to a full GUI environment. You should feel free to explore this environment and use it. However, this handout focuses on how to do things in the command line environment, which is how you do things with ssh, and which is necessary for some purposes (such as compiling and running your programs). As you use unfamiliar commands, it is often helpful to type the command followed by `–help` to learn more.

Getting a command line window in the GUI
If there is a terminal or console window on your screen, you can activate it by simply clicking in it. You can start a new terminal window by selecting Terminal Emulator from the Applications menu at the top left of your screen or by clicking on the Terminal Emulator icon, which will be the second from the left in the bar in the middle of the bottom of the screen.

Leaving

When leaving, you may simply disconnect, which will leave your session in place for a limited period of time. If you do not plan to reconnect soon, it is probably better to log out. Please do not try to shut down. These are shared machines.

Directory Structure

Linux maintains a hierarchical file structure consisting of a root directory and several layers of subdirectories. These are much like the folders in the Windows and Macintosh operating systems. Each directory contains zero or more files and subdirectories. The root directory is always named **`/`** and referred to by this character.

At any point in time, each user is associated with a single directory called the *working (or current) directory*. You should think of this as the folder you are currently "in": the place where any commands are run. Immediately after logging in to a Linux system, a user is associated with a *home directory*. A possible home directory would be named **`/home/ad.ilstu.edu/tpaine`**, where "tpaine" is the person's login name. Linux uses the tilde (**`~`**) as a shortcut for your home directory. Note that Linux uses a forward slash to separate directory names, unlike Windows, which uses a backslash. Note that home directories on our system are typically `/home/ad.ilstu.edu/`*`ulid`*. This naming scheme is a result of the mechanism that allows us to use our standard ISU accounts to log in to these systems.

Naming Files and Directories

Almost any character can be used in the name of a file or directory, but try to use only the following characters: letters (upper or lower case), digits, underscore, hyphen, and period. Please do **not** use spaces in file and directory names, because they do cause some annoyance when working from the command line (as you should and I always will).

Either an absolute or a relative pathname may be used whenever it is necessary to specify the name of a file or a directory.

To write an absolute pathname, start with the root directory and list, in order, all of the directories that contain the desired object. Place the name of the desired object at the end of the list and separate all names from one another with the **`/`** character. For example, suppose that a file named **`liberty`** is in the home directory of user **`tpaine`**. The absolute pathname of this file would be **`/home/ad.ilstu.edu/tpaine/liberty`**. Remember never to use **`/`** to begin a relative pathname; you only use **`/`** to begin an absolute pathname.

A relative pathname can be used to specify where a file is in relation to the working directory.

Example 1
Suppose that a file named **`barney`** is in the working directory.
The relative name for this file would be **`barney`**.

Example 2
Suppose that a file named **`dino`** is in a directory named **`pets`** which is contained in the working directory.
The relative name for this file would be **`pets/dino`**.

`.` and **`..`**
The period and the double period are important symbols in dealing with directories. The single period refers to the working directory. This means that **`./pets/dino`** is equivalent to **`pets/dino`**. The double period refers to the parent of the working directory.

Editing files

I highly recommend that you edit files using **Visual Studio Code**, which can be accessed from the command line by typing **`code`**. To edit a specific file, you can type **`code`** ***`filename`***. You can open a whole folder with the same format (e.g. open your current directory with **`code .`**). This will be your primary approach. You can also access VS Code from the Applications menu under Development.

Note that you may add extensions to VSCode. Make sure to add the Live Share extension right away, since that will be very helpful for completing group homework and getting program help from me. Link: Live Share - Visual Studio Marketplace.

Note that VSCode also has a built-in terminal that can be used just like the terminal windows you create in the GUI environment.

Commands for getting around

`pwd`
Stands for print working directory. Displays the absolute pathname for the current directory. If I were in my home directory, the result of pwd would be

```
      /home/ad.ilstu.edu/mecalif
```

`cd`
Stands for change directory. Use this command to switch directories. It typically takes one argument (the directory to move to). With no argument, it makes the user's home directory the working directory.

Examples:
Note: On our system, the prompt at which you type will consist of
`systemName``:`*`workingDirectory`*`$`
With the directory path using the tilde shortcut. So if you are in your home directory on bur, your prompt would be

```
       bur:~$
```

If you switched to your Documents directory, your prompt would change to

```
       bur:~/Documents$
```

If you were in my public directory on ash, your prompt would be

```
       ash:/home/ad.ilstu.edu/mecalif/public$
```

Because these prompts can get quite long, some of my examples will use just `$` for the prompt the system displays. The command to type is what is in bold after the `$`.

```
$ cd pets
```

Will change the working directory to pets, a subdirectory of the current working directory

```
$ cd ../program2
```

Will change the working directory to program2, which must be a "sister" directory of the current working directory (i.e. the two directories have the same parent directory).

```
$ cd
```

Will change the working directory to the user's home directory—on our system typically
`/home/ad.ilstu.edu/`*`ulid`*.

`ls`
Use this command to list the files in the working directory. You can give it a directory name to see the contents of a particular directory. The ls command has several useful options. The most used of these is -l, which gives a "long" file listing with more detailed information about each file.

Commands for dealing with directories

`mkdir`
Stands for make directory. Use to create a new directory within the working directory. Takes one argument: the name of the new directory.
Example:

```
ash:~$ mkdir IT279
ash:~$ cd IT279
```

```
ash:~/IT279$ mkdir prog1
```
This would create a directory named IT 279 and a directory inside it named prog1.

`rmdir`

Stands for remove directory. Use to delete directories. You must provide the name of the directory to be deleted. Can be given multiple directories at once. **Note:** only empty directories can be deleted using rmdir.

Dealing with files

`*`

When specifying file names, it's often the case that you want to refer to all of the files in a directory or all of the files of a particular type. This can be done using *. This is a wildcard operator, meaning you can also use this to represent any characters in a file name (e.g. File names starting with "lin" and ending in ".txt" = lin*.txt).

`cp`

Stands for copy. Takes two arguments: the file to be copied and the name or location of the new copy. If the second argument is the name of a directory, a file with the same name as the original is placed in the directory. Otherwise, the second argument is the name of the new file. Can be used to copy multiple files if the last argument is a directory. Note that you do not need to be in any particular directory to do a copy; you must simply specify enough information about the file to locate it.

Example:
```
$ cp /home/ad.ilstu.edu/mecalif/public/it279/Program1/* .
```

This command will copy all files from the Program1 directory to the working directory. Note that the working directory has been specified using a single period and that there is a space between the asterisk and the period.

`mv`

Stands for move. Can be used to rename a file or to move it from one directory to another. Like cp, takes two arguments: the name of the file to be moved and the new name or location of the file. Can be used to move multiple files if the last argument is a directory.

Example:
```
$ mv *.h *.cpp progs
```
This command will move all files in the working directory that end in .h or .cpp into the subdirectory named progs.

`rm`

Stands for remove. Can be used to delete files. Be careful, since this is not an undoable action. You can use the option -i to have it confirm each deletion (tedious at times, but safer). You can also use the option -r to recursively remove directories and their contents. Again, not undoable.

`zip`

After you write your programs for class, you will need to submit them, which will require zipping the files. For this course, you must zip **only** the source code files that you are required to submit. The command for this is zip. The initial parameter will the zip file, and subsequent parameters will be the files that will go into the zip. Example:

```
$ zip prog1.zip MyClass.h MyClass.cpp MyProg.cpp
```

This will create the file prog1.zip containing the three source code files. Be very careful with this command; it can be easy to destroy one of your files by forgetting to put the zip file name first.

`unzip`

Unzipping the file is pretty trivial. You're just going to use the unzip command on it. This does require that the file name end in .zip. Example:

```
$ unzip prog1.zip
```

Compiling C++ programs

To compile a C++ program, use the command **`g++`** followed by **all** of the .cpp files in the program. Note that the .h files are not included on the command line, only the .cpp files. When working on a program with multiple files, you may find it useful to create a makefile containing instructions on how to build the program. You can find a sample makefile in /home/ad.ilstu.edu/mecalif/public/it279/CodeExamples/stack_queue_new. I will be glad to sit down with you and help you create your first makefiles and figure out how **make** works. You can also look at the **make** manual page by typing `man make` for information.

If you're writing programs of any size at all, it's a good idea to create a directory for each program. This makes compilation very easy, since the * works here, too.

Example:

```
$ g++ *.cpp
```

This will correctly compile your program if all of the .cpp files in the working directory are part of your program.

The executable produced by the command above is named **a.out**. This is the default name for any C/C++ program you compile. You can then run the program by typing **`./a.out`** at the command prompt. However, you can specify a name for the program by using the –o option (the o stands for output). You must use the ./ before the executable name so that the operating system knows to check only your current directory.

Example:
```
$ g++ -o myprog *.cpp
```
Instead of creating an executable program named a.out, this creates an executable program named myprog. You would then run the program as follows:
```
$ ./myprog
```

Running and Testing Your C++ programs

Essential Commands and Concepts

This section is about a variety of commands and concepts that will be helpful to you as you complete your assignments and test your programs this semester.

Input and Output Redirection

Linux allows us to redirect input and output of commands. This can be useful in a variety of ways. It is especially useful for testing your programs. It allows programs to be flexible because they can work on standard input or write to standard output while allowing the user to store the data in files. We can even string commands together, sending the output of one command to the next command as input.

`<` redirects the console input of a command to come from the following file. Often useful in running programs many times with the same input. I regularly use this technique for grading. You will also use this to do some testing of programs in this course.

`>` redirects the console output a command (printing) to go to the following file. This can be useful for storing the output of a program in a file for later perusal.

We often use both input and output redirection at the same time, but this is not necessary: you can use just one of the two, depending on the need and the specifics of the program.

`cat`

Lists the contents of the file argument to standard output (typically the screen, just as in your experience on Windows). If multiple filenames are provided, each file is listed in the order in which the files are given to the command.

`diff`

Allows you to compare two files to see if they're the same. Very useful for testing programs. Simply type `diff` *`filename1 filename2`*. You are **expected** to use **`diff`** to check your programs against provided sample output before submitting. This is **exactly** how your program will be evaluated for accuracy/correctness against sample output files, and against other test cases you may not be provided.

`time`

Can be used to determine the time programs take to run. To use this command, you simply type "time" followed by the command to run the program. The time command gives you three numbers: the "real" time that passed in the world while the program ran (also called the "wall clock" time), the user cpu time, and the system cpu time. The number of primary interest to you will be the sum of the second and third values. The real time is irrelevant, in general, because it varies with the load on the machine. You will be required to use the time command for one of your homework assignments and for your first paper.

Demo Exercise

In this section, I am walking you through trying out the various commands as you would use them in creating and testing your programs this semester. I **strongly** encourage you to work through this and **do** everything. If you are having trouble using these commands, please see the explanations in this document to help you. That's what they're there for! If that is not enough, I will be available through Teams.

- Log in to one of the Linux machines following the directions above.
- Open a terminal window. Notice that you are in your home directory.
- Type `pwd` and press the enter key. You will see the full path to your home directory, since that is your current working directory.
- It will be helpful to have a directory for all of your IT 279 work, so use `mkdir` to create a directory called `IT279` or something similar (remember that you don't want spaces in your directory and file names).
- Use `cd` to switch to your `IT279` directory and create another directory within it for this exercise called `TryIt`. I strongly recommend creating a directory for each different assignment in the course. Sometimes you'll find it convenient to have multiple directories, particularly for programming assignments where you write multiple programs.
- Switch into the `TryIt` directory using cd.
- Now we want to copy all of the files from a directory in my public space (/home/ad.ilstu.edu/mecalif/public/it279/CodeExamples/LinuxPractice) into your current working directory. You can do that with the command:

```
cp /home/ad.ilstu.edu/mecalif/public/it279/CodeExamples/LinuxPractice/* .
```

- Use `ls` to see what you have.
- Use VS Code to look at `CountNums.cpp`. Then compile it using `g++`.
- Use ls -l to see the permissions and other details of your files. Note that the files you copied have read-write permissions only, but the file you created with g++ is executable as well.
- Run the program with command line parameter 10 and put in just a few values manually (say 5).
- Obviously, this is not the ideal way to handle a lot of values so we're going to use input and output redirection to run our program more conveniently.
- You have two sample input and output files. Let's use redirection to run the program again with a parameter of 100, using the `test_data1` file as input and storing our output in `outfile1`. Assuming you used the default program name of `a.out`, that command is:

```
./a.out 100 < test_data1 > outfile1
```

- To see what we have in `outfile1`, we have several options. Of course we could use VS Code, but there are simpler options described below. To just list the whole file to the terminal, we could use `cat`. To page through the file in a readonly mode, we could use `more` or `less` (see below). Go ahead and try one or both of these options yourself.
- In addition to just looking at our output, it's often helpful to compare it to the correct output that your teacher or her grader has helpfully provided. To do that, we have two options. The simpler one is to use the diff command. The provided `sample_outfile1` is correct output for the program you ran. Use diff to compare your output to the expected output:

  ```
  diff sample_outfile1 outfile1
  ```
- You should see no output, indicating that the two files are the same. I encourage you to run the program with the sample input files, but using a parameter of 50 instead of 100, then diff your new output with the sample. You should see differences listed in the terminal window.
  - The arrow direction in the diff output will indicate which file the line came from. For example, "< test" is a line with "test" on it that was in the first file and not the second file.
- You can also compare files using VS Code. To do so: select two files using Ctrl+Click, right click one of the files, select "Compare Files". This has the advantage of allowing you to see them side by side with the differences highlighted, which will help you determine the reason for the difference at times. However, diff is a great starting place, since it is much faster. Use VS Code Compare in the same two cases as you did with `diff`, to see how the result is formatted in VS Code.
- Another thing you will need to do with programs is to time them. We're very interested in program performance and comparing programs to one another in terms of execution time on similar data as well as comparing actual performance with theoretically expected performance. To do this we will use the time command.
- I have given you a few data files of different sizes. Try running each of them using the time command and look at the differences. Recall that you should add the usr+sys times to get your most accurate run time. Sample command:

  ```
  time ./a.out 100 < test_data1 > outfile1
  ```
- Let's now clean up after ourselves, since we are finished running the program. Delete the files in your TryIt directory. You can do this using:

  ```
  rm *
  ```
- Switch to your IT279 directory using:

  ```
  cd ..
  ```
- Now delete the TryIt directory with:

  ```
  rmdir TryIt
  ```

You have now completed the demo! Congratulations, and I hope that you were able to get good value out of this practice. The code we used will remain available to you in my public area.

File Permissions

By default, your created files will be given permissions for you to read and write them only, but not execute. Nobody else will have any permissions for your files. To see the file permissions for the files in your current directory, type `ls -l`. Permissions are broken up into 3 groups: Yourself, Your Group, Everyone else. Generally, if you don't know what group you are in, just use the same permission level as the "everyone else" group. There are also 3 kinds of file permissions: reading (r), writing (w), and executing (x).

In the below example, you'll see I have one file named HelloWorld. On the very left there are 10 file information flags which are bolded. The first flag will identify special characteristics of a name, such as it being a directory with the character "d". The next 9 flags are used for file permissions. In this example, I have all access (read, write, and execute) to HelloWorld, the group I am a part of has read and write access only, and everyone else has read access only.

```
-rwxrw-r-- 1 mecalif domain users 576 Jan  5 15:29 HelloWorld
```

Giving Others Access to Your Files

In this class you will sometimes need to give me access to your code files, usually when you ask for help with a bug in your program. **Note that you are not permitted to share your programs with other classmates, so be careful with your file permissions.** There are two ways to manage your file permissions, using the chmod command or using a script in my public directory (which makes use of chmod) covered on the next page.

`chmod`

This command is used to change the file permissions of one or more files that you specify in the command line arguments. The general format of executing the command is `chmod <permissions> <1 or more filenames>`. The permissions are most easily done using decimal numbers. Each of 3 decimal numbers from 0-7 will be used to represent the 3 permission user groups (yourself, your group, everyone else). If you want to allow read and write permissions only for a permission user group, you would use:

(read)(write)(execute) → (1)(1)(0) → 110 = 6.

Combining three of these decimal numbers with the accesses you want, will give you the number to use in `chmod`.

In the above example of the `ls -l` command, I would use `chmod 764 HelloWorld` to set those permissions. Note that for someone to have access to a file, they must be able to have read and execute access in every directory above it. This must be done manually if using the chmod command. Please make sure to revert any file permissions changes you make once you are done.

In general, you will tend to use chmod 644 filename to make text files (such as source code files) visible to others, and chmod 755 directoryname to make directories accessible to others.

`accessPerms.sh Script`

This script has been created to facilitate allowing access to one or more files inside your **current directory**. The advantage of using this script is that it is easier since it takes care of the nested directory permissions for you. The script is located at `/home/ad.ilstu.edu/mecalif/public/accessPerms.sh`. This script takes one parameter at the beginning to specify whether to open or close the directory stack following by one or more additional parameters that will be file names. This is how I would run the script to give access to my HelloWorld file. Alternatively, you can copy the script to your directory to save some extra typing. Either way, make sure that you are running it while in the directory with the files in it.

```
$ /home/ad.ilstu.edu/mecalif/public/accessPerms.sh --open HelloWorld
```

IMPORTANT: Until you run the appropriate closing command, **ANYONE** will have access to the files that you have specified. The script will print out directions on how to re-run the script to close your directory stack and restrict permissions again. You should use the `--close` operation specified in those directions to revert the permissions changes as soon as you do not need your files to be public.

Other useful details and commands

Additional redirection example:

`$ cat file1 file2 > file3`

This will create a new file named file3 with the contents of file1 and file2.

`wc`

Stands for word count. Given a file name, it provides the number of lines, words, and characters in the file. Given multiple file names, it provides the data for each file, followed by a summary line with totals. This is useful for things like determining that your sort files produced the correct number of values in the output.

`|` is called a pipe. It is used to pass the output of one command into the input of another command (through the pipe).

Example:

`$ grep abc testfile | wc`

The `grep` command is executed, producing all of the lines in testfile that contain the string “abc”; then those lines are passed as input to `wc`, which displays a count of the number of lines as well as the number of words and characters in those lines.

`man`

Stands for manual. When followed by a command, will show the manual page for the command. The man pages are somewhat challenging to read at first, but are useful for

finding out all the different options for a command or remembering exactly how to use a particular command.
Example:
```
$ man sort
```
Will give information about the sort command and the various options that are possible for modifying the sorting.

`apropos`
A useful command for finding a command/man page if you're not sure of the name of the command you are looking for.

`grep`
This command is used for searching in files using a *regular expression*. See the man page for more information. Linux uses limited regular expressions for understanding command line arguments, which is why * works as it does.

`sort`
Can be used to sort files. Has several useful options. See the man page for details.

`more` or **`less`**
These commands can be used to browse text files a page at a time. `less` is a little more sophisticated than `more`. Both allow
searching by typing **`/`** followed by the search pattern
searching for the same pattern again by typing **`n`**
moving a line at a time forward by pressing the enter key
moving a page at a time forward by pressing the space bar
moving a page at a time backward by typing **`b`**
quitting by typing **`q`**

These are sometimes useful for looking through a file that is too large for VS Code to work with efficiently.

`ctrl-c`
While you should always program very carefully, even the best of us occasionally manage to create a runaway program (infinite loops). If you need to make your program stop, the first thing to do is to type ctrl-c. That will almost always kill the program. If that does not work, open another terminal window or create another connection to the **same** machine and use the **`ps`** command to find out what the *process id number* is for your runaway program. You may have to use the `-a` option to find your program. Once you know the process id, you can use the **`kill`** command to stop your program. You'll use the -9 option for the kill command to guarantee that the process stops.

Example:
Suppose that you have created a program named "fred" that is stuck in a loop and ctrl-c didn't work. You open another window and try the ***ps*** *command.*

```
$ ps
   PID TTY      TIME CMD
 11327 pts/9    0:00 tcsh
```

Since your program didn't show up, you try ***ps -a***

```
$ ps -a
   PID TTY      TIME CMD
  1048 pts/3    0:02 dtsessio
  1343 pts/9    0:00 ps
 11327 pts/9    0:00 tcsh
 11468 pts/6    0:13 fred
 11598 pts/3    0:00 more
```

Here we see that the command fred has process id 11468, so we can now perform the kill command.

```
$ kill -9 11468
```

Using ReggieNet to Submit Assignments

Open a browser window from the icon at the bottom of the screen or from the Applications menu. You will find ReggieNet at reggienet.illinoisstate.edu.