Using Your Linux Account

Original text by Mattox Beckman, Jan 27, 2014

Modified by J. Sasaki, Feb 5, 2016

Table of Contents

1 Introduction

Welcome!

As part of this course, you were given an account on the Linux machine `alpha.cs.iit.edu` (`alpha` for short). This account has been provided to you courtesy of the student chapter of ACM, the Association of Computing Machinery. The purpose of this document is to give you the information you need to use your account effectively to develop your programs and run them.

If you don't want to use `alpha` for initial development of your programs, you don't have to; you are welcome to use your own machine. If your home machine is a Linux or Mac OS machine, then the commands discussed in this document should all behave the same way on your machine as on `alpha`[1]. (You may need to upload a copy of the GNU C compiler, "`gcc`", however.) If your home machine runs Windows, then you will need to find a C compiler. They tend to come with an IDE (Integrated Development Environment). For what we're doing in this class, an IDE is likely to be overkill, but you're welcome to use one if you want.

However you develop C programs on your machine, once you finish testing and debugging one on your machine, you will need to upload it to `alpha` to test it there, because that's where we will test them for grading purposes. Buggy programs may behave differently on different machines, but in general, programs that run correctly on your home machine should run correctly on `alpha` so long as you avoid non-standard libraries like `conio.h` (console IO on Windows).

2 First Login

2.1 Initial Account Information

If you had an account on `alpha` in the past, it may have expired; you'll need to get in touch with the `alpha` administrator to reactivate it. If you're a new user, then when you first get an account on the machine, you will get an email like this one. (It might be in your spam mailbox.)

```
This email is to inform you of your linux account.
The machine is named alpha.cs.iit.edu.
Your username is USERNAME.

Your password will arrive in a message just after or before this one.
The subject will be 'alpha', and the body of the message will be
the password, a string of 8 hexadecimal digits.

That's not the most secure way to deliver a password, so you will need
12to log onto your account and change it within the next few days, or
else the account will be suspended.

Enjoy!

- alpha account creation program
```

The word `USERNAME` will be replaced by something that presumably is related to your actual name. If you have an ITT email address, then your username is most likely the same. You will then get another email with the subject "`alpha`". In it will be something like this:

```
dea0667c
```

This is your initial password. The account creation program generated it randomly. It will only be good for one use.

2.2 Secure Shell

To log on, you will need to use the *secure shell protocol,* also called SSH. "SSH" stands for "**Secure Shell**". (A **shell** is a command-line-based interface to a unix-like operating system like Linux or Mac OS.) Using an SSH client, you can remotely log in to `alpha` to edit and run your programs.

---

[1] More or less; details like the exact path to your home directory or what shell prompts you see may differ.

If you're working on another unix machine[2], you can use the command line. Start up a terminal program and run the command

```
ssh username@ alpha.cs.iit.edu
```

For Windows users, we recommend you install the SSH client PuTTY (http://www.putty.org/). Set it up to connect to `alpha.cs.iit.edu` using SSH.

The very first time you connect to a machine using SSH, you will get a message about the server host's key not being known. Assuming you trust that you've connected to `alpha` and not some machine pretending to be `alpha`, go ahead and accept the host key.[3] It will be stored and you won't see this message in future connections.

2.3 First Connection

Whether you try the `ssh` command or use `PuTTY`, you will get one of two **prompts**. (A prompt is something the computer types out to tell you that it's ready for input.) You might get a prompt that says

```
login:
```

or perhaps

```
alpha login:
```

If you get that, type in your username and hit enter. After that, the second prompt will show up. For many of you, it will go straight to this one without asking for a username. It's called the password prompt, and will look like this:

```
Password:
```

You then need to type in your password and hit enter. Note: you will not see any feedback at all until you hit enter. Web sites like to put asterisks to show your typing; unix systems show nothing at all. It is slightly more secure this way, but a little unnerving the first couple of times. If you make a mistake typing the password, the ctl-U key is a shortcut to erase it so you can start over. An important note on etiquette: it is considered polite—in the same way that not sneezing into the food at the cafeteria is considered polite—to turn your head if someone else is typing in a password.

If all went well, you are now logged in. If something went wrong (a typo, or perhaps your account wasn't created yet), then you will get the Username prompt again. If this keeps happening, ask the instructor for help. If you keep typing in the wrong password (more than about 10 times) the security software starts to take notice and could block your machine from connecting.

2.4 Next Step: Getting A Real Password

When you connect for the first time, you will need to change your password. The screen will look something like this:

---

[2] From here on, I'll use "unix" to mean any unix-like operating system. You're most likely to encounter Linux or Mac OS, but there are other systems, such as Solaris and FreeBSD.

[3] If you're on campus connected via IIT-Secure WiFi or you're off campus using the VPN, you're almost certainly safe. If you're using a free public WiFi connection at a coffee house in (fill in various countries), you might be a bit leery.

```
1     username@ alpha.cs.iit.edu's password:
2     You are required to change your password immediately (root enforced)
3     WARNING: Your password has expired.
4     You must change your password now and login again
5     Changing password for username
6     (current) UNIX password:
7     Enter new UNIX password:
8     Retype new UNIX password:
9     passwd: password updated successfully
10    Connection to alpha.cs.iit.edu closed.
```

You will type in a password four times. Again, you will not see the cursor move or get any feedback as you type in your password. This is standard behavior on unix systems. The first time you type in your password is on line 1. This is for your initial login. The password is set for single-use, so the machine will automatically run the password change program. This program requires you to type in your old password, which you will do on line 7. After you hit enter, you will be prompted to create a new password. Once you have entered that, you will have to type it again to verify that there were no mistakes. Once that is done, you will be rudely logged out. Simply log in again, this time with your new password.

2.5 Your First Real Login

The first time you log in with your new password, you will get the welcome screen. It will have a bunch of stuff on it that you can ignore for the moment we'll explain it more later. It should look something like this.

```
Last login: Fri Sep 5 09:41:28 2008 from
ads1-68-20-176-127.dsl.chcgil.ameritech.net
KeyChain 2.6.8; http://www.gentoo.org/proj/en/keychain/
Copyright 2002-2004 Gentoo Foundation; Distributed under the GPL

Initializing /home/students/all/username/.keychain/alpha.cs.iit.edu-sh
file...
Initializing /home/students/all/username/.keychain/alpha.cs.iit.edu-csh
file...
Initializing /home/students/all/username/.keychain/alpha.cs.iit.edu-fish
file...
Starting ssh-agent
Adding 1 ssh key(s)...
Identity added: /home/students/all/username/.ssh/id_rsa
(/home/students/all/username/.ssh/id_rsa)
username@alpha ~ $
```

2.6 You're in!

You now have access to your Linux account. The next three sections tell you what you need to know to make good use of it. Section 3 introduces some basic Linux commands. Feel free to skip this section if you are already familiar with unix. Section 4 addresses compiling and running C programs.

3 Basic Unix Commands

An unfortunate thing about unix is that you have to learn quite a bit at the very beginning in order to get started. But once you have gained some proficiency, you will be very productive.

Recall that a shell is a command-line-based interface to a unix operating system. Basically, you type commands into a program that interprets them to manipulate files and run programs (yours or unix's).There are many different shells you can choose from to meet your needs. The one on `alpha` is called `bash`. In the previous section you saw how to log on. At the bottom of the screen you saw something like this:

```
username@alpha ~ $
```

This line is called the **prompt**. Prompts are highly customizable; the one you see here shows the name of the user (*username* in this case) and the short name of the machine (`alpha`). (It also shows your current location within the file system, using the tilde; see Section 3.2 below.)

The last character of the prompt ("$", above) indicates an important piece of information. If it is a `%` or `$`, then you are logged into a user account. If it is a #, then you are logged into the super-user account (also known as the root account). A super-user has full administrative rights on the machine. You should not expect to get to see this prompt on `alpha`!

3.1 The Echo Command; Standard I/O, and Simple cat Commands

3.1.1 Echo

Probably the simplest command is `echo`. You type `echo`, space, some characters, and carriage return. The command prints out the characters. Once the shell has executed a command, it prompts you for the next command. (Below, your input is shown *in italics*.)

```
username@alpha ~ $ echo hello world!
hello world!
username@alpha ~ $
```

Note `echo` takes just one line of data, so if you want to print `hello` and `world!` on two different lines, you need two `echo` commands. (Hope you don't mind the prompt between the two lines.)

```
username@alpha ~ $ echo hello
hello
username@alpha ~ $ world!
world!: Command not found
username@alpha ~ $ echo hello
hello
username@alpha ~ $ echo world!
world!
```

Above, the error message about the line `world!` indicates that the shell thought you meant to run a program or command named "`world!`", but it couldn't find one.

3.1.2 Stdin, stdout, cat, and redirecting stdout

The echo command prints its output to the "**standard output**" ("**stdout**" for short) which by default is the screen. You can, however, "**redirect**" the standard output to a file to save it. You redirect stdout using the ">" character followed by a filename.

```
username@alpha ~ $ echo hello world! > hello.txt
username@alpha ~ $
```

You can't see the contents of a file using `echo` *filename* — that just prints out the filename. To see the contents of a file, you can print it to the screen using the `cat` command (short for

"concatenate" — we'll see that use in a bit). You say `cat` *filename* and it prints that file's contents to standard output.

```
username@alpha ~ $ echo hello world! > hello.txt
username@alpha ~ $ echo hello.txt
hello.txt
username@alpha ~ $ cat hello.txt
hello world!
username@alpha ~ $
```

Since `cat` prints to standard output, you can redirect that output to a file. Why do this? One reason is that it's a simple way to type in some multi-line text to a file. Without an input filename, `cat` takes the "**standard input**" and copies it to standard output. By default, the standard input is the keyboard. To signal the end of your input, use ctl-D (control-D) on the line after your last line of actual input. (Below, I'm showing where you type the `^D`, but the shell might not display it.)

```
username@alpha ~ $ cat > myfile.txt
my file is a very very very fine file
with two cats in the yard
^D
username@alpha ~ $ cat myfile.txt
my file is a very very very fine file
with two cats in the yard
username@alpha ~ $
```

You can list more than one file; `cat` *file1 file2* will copy first *file1* and then *file2* to stdout. If you redirect stdout to a *file3*, then it will contain the concatenation of the two files; otherwise stdout goes to the screen:

```
username@alpha ~ $ cat hello.txt myfile.txt
hello there!
my file is a very very very fine file
with two cats in the yard
username@alpha ~ $
```

3.2 The Current Directory; pwd, ls, mkdir, mv file, and cd

3.2.1 pwd; the simplest ls command

Unix operating systems organize files using **directories**; a directory can contain files and/or subdirectories. When you type commands in to the shell, the commands are taken relative to the "**current**" directory. Basically, you're always "in" or "at" a directory.. The `pwd` ("print working directory") command prints out the name of the current directory. The `ls` (**list directory**) command prints the names of all the files and directories inside the current directory. (Depending on what you've been doing on `alpha`, the result of your `ls` may vary.)

```
username@alpha ~ $ pwd
/home/students/all/username
username@alpha ~ $ ls
hello.txt  myfile.txt
```

In the output above, the user happens to be in the directory `/home/students/all/username`. It is really five directories nested within each other. The top-level directory is called simply `/` (pronounced "slash" or "root"). Within the root directory is a subdirectory called `home`,

which contains a subdirectory called `students`, which contains a subdirectory called `all`, which in turn contains a subdirectory called `username`.

3.2.2 Making directories; moving files into directories; ls directory

You can get in on the fun by making your own directories using the `mkdir` command.

```
username@alpha ~ $ mkdir mydir
username@alpha ~ $ ls
hello.txt mydir    myfile.txt
```

Newly-created directories are empty. You can move a file from the current directory to a subdirectory using the `mv` ("**move**") command: `mv` *filename directory* moves the named file to be inside the named directory. By default, `ls` won't show you the contents of any of your subdirectories. One way around this is to use `ls` *directory*, which lists the contents of the named directory.

```
username@alpha ~ $ mv hello.txt mydir
username@alpha ~ $ ls
mydir myfile.txt
username@alpha ~ $ ls mydir
hello.txt
```

To access a file in a subdirectory, you use *directory/ filename*

```
username@alpha ~ $ cat hello.txt
cat: hello.txt: No such file or directory
username@alpha ~ $ cat mydir/hello.txt
hello world!
```

3.2.3 Changing directories; current directory named in prompt

To change directories, you use the command `cd` *directory*. Note that after the `cd`, the `pwd`, `ls`, and `cat` commands all do their work relative to the new directory

```
username@alpha ~ $ cd mydir
username@alpha ~/mydir $ pwd
/home/students/all/username/mydir
username@alpha ~/mydir $ ls
hello.txt
username@alpha ~/mydir $ cat hello.txt
hello world!
username@alpha ~/mydir $
```

Also notice how the prompt changes as you switch directories. On `alpha`, the default prompt includes the name of the current working directory. The special filename "~" (tilde) is unix shorthand for the name of your **home directory**, the main directory in which your files are placed. When our ssh session started out, we began in the home directory (hence the tilde in the prompts); when we changed to `mydir`, the prompt changed to `~/mydir` to show that that's where we were. If you type `cd ~`, that will bring you back to your home directory. As a shortcut, `cd` by itself will do the same thing.

```
username@alpha ~/mydir $ cd
username@alpha ~ $
```

3.3 Customizing ls: ls -a -F -l; ls wildcard

You can customize the behavior of `ls` (and in fact, just about every command) by using switches or command-line options. A switch begins with one or two dashes (i.e., hyphens). A one-dash switch usually has just one letter after it, while two-dash switches are more verbose. Here are some one-dash switches for `ls` and what they do.

3.3.1 ls -F

The `-F` flag prints a `*` after executable files and a `/` after directory names. Flags are generally case-sensitive, so don't use `ls -f`

```
username@alpha ~ $ ls
hello.txt mydir   myfile.txt
username@alpha ~ $ ls -F
hello.txt mydir/  myfile.txt
```

The `-F` flag makes it very easy to tell which files are directories, etc.

3.3.2 ls -a

Filenames that begin with `.` are usually hidden when you do an `ls`. The `-a` flag ("**all files**") tells `ls` to show everything.

```
username@alpha ~ $ ls -a
.     ..  .bashrc  .bash_profile    hello.txt    mydir    myfile.txt
```

(Your exact output may vary from the output above.)

By the way, you can combine flags; you can use `ls -a -F` to combine both features. The `ls` command also lets you combine the flags textually into `ls -aF`:

```
username@alpha ~ $ ls -aF
./    ../ .bashrc  .bash_profile    hello.txt    mydir/   myfile.txt
```

The `ls -a` outputs above include are four hidden files. The `.` and `..` names indicate the current and parent directories, respectively.

With the exception of `.` and `..`, most files that begin with `.` are configuration files of some sort. (They often end in "`rc`" for "resource"; `.bashrc` is an example.) In unix, configuration information for different programs generally appears in different files. While this does mean you have to hunt for the proper file to configure a program (usually, for a program *foo*, the user configuration file will be called `.foorc`), you avoid the unpleasantness that occurs in Windows when something corrupts the registry.

3.3.3 ls -l

The `-l` (letter `l`) flag means **long output**. The file and directory names are listed one per line and extra information is included about each one.

```
username@alpha ~ $ ls -l
total 8
drwx------ 2 username users 4096 Jan 23 19:52 mydir
-rw------- 1 username users  127 Jan 23 19:36 myfile.txt
```

Let's break down this output. The first line contains the total size of the files in the directory.

The second line of the `ls -l` breaks down into

`drwx------`	`d` for "this is a directory", plus info on who can read/write/execute this file[4]
`2`	The number of links to this file (a file can be a member of multiple directories)
`username`	Name of user who owns the file
`users`	Name of group that this file is in
`4096`	Size of file
`Jan 23 19:52`	Date and time of last modification
`mydir`	Name of file

The third line of the `ls -l output` indicates that `myfile.txt` is not a directory; the permissions, size, and modification date/time are different from those for `mydir`.

If you don't want to list everything, you can ask to list specific files:

```
username@alpha ~ $ ls -l myfile.txt
-rw------- 1 username users 127 Jan 23 19:36 myfile.txt
```

An `ls` of a directory lists the contents of that directory.

```
username@alpha ~ $ ls mydir
hello.txt
username@alpha ~ $ ls -l mydir
total 4
-rw------- 1 username users 13 Jan 23 19:36 hello.txt
username@alpha ~ $
```

3.3.4 Wildcards

You can also use *wildcards.* For example, to get all the files ending in `txt` you can use `*.txt`:

```
username@alpha ~ $ ls
mydir  myfile.txt
username@alpha ~ $ ls *.txt
myfile.txt
username@alpha ~ $
```

You can use just `*` to mean all (non-hidden) files; note the difference between `ls` and `ls *` (if the `*` includes any directories, then their contents are printed out).

```
username@alpha ~ $ ls -F
mydir/  myfile.txt
username@alpha ~ $ ls *
mydir/  myfile.txt

mydir:
hello.txt
username@alpha ~ $
```

[4] The format of this is complicated and I'm not going to try to explain it here. Use the "`man ls`" command to read more about it some time when you have some extra time.

3.4 Copying, Deleting, and Renaming Files and Directories

3.4.1 Copying a file

To copy a file into an existing directory, use `cp` *filename directory*.[5] To make a duplicate copy of a file, use `cp` *filename nameOfNewCopy*

```
username@alpha ~ $ ls mydir
hello.txt
username@alpha ~ $ cp myfile.txt mydir
username@alpha ~ $ ls mydir
hello.txt myfile.txt
username@alpha ~ $ cp myfile.txt mycopy.txt
username@alpha ~ $ ls -F
mycopy.txt mydir/ myfile.txt
```

You can also use the `cp` command to make a duplicate copy of a directory and all its contents: We won't discuss that here. (Look up the `-r` flag of `cp` if you're interested.)

3.4.2 Renaming a file or directory; mv and mv -i

To rename a file or a directory, use the move command: `mv` *filename newname* or `mv` *directory newname*

```
username@alpha ~ $ mv myfile.txt file.txt
username@alpha ~ $ mv mydir d
username@alpha ~ $ ls -F
d/    file.txt    mycopy.txt
```

For a rename, make sure the new name doesn't already exist. If *filename2* already exists, then `mv` *filename filename2* will **replace** *filename2* by *filename*. (Hope you kept a copy of the old *filename2*.) If you want `mv` to make sure you mean to do a replace, use the `-i` (interactive) flag:

```
username@alpha ~ $ ls
d     file.txt    mycopy.txt
username@alpha ~ $ mv -i file.txt mycopy.txt
mv: overwrite 'mycopy.txt'?
```

You can enter `y` or `n` at the prompt as appropriate[6].

To rename a directory, again make sure that the new name isn't being used. If the target name is an existing file, you'll get an error message.

```
username@alpha ~ $ mv d file.txt
mv: cannot overwrite non-directory 'file.txt' with directory 'd'
```

If the target name is an already-existing directory, then the first directory will be **moved into** the second directory — it will become a subdirectory of the target.

```
username@alpha ~ $ mkdir d2
username@alpha ~ $ mv d2 d
```

---

[5] You can actually move a bunch of things into a directory: `mv` *file1 file2 file3 directory*, for example, instead of three individual `mv` commands.

[6] It's possible that someone (even you, perhaps) has changed the default configuration so that `mv` always behaves like `mv -i` (see Aliases below). In that case, use `mv -f` (force) to avoid the overwrite prompt.

```
username@alpha ~ $ ls -F d
d2/  hello.txt    myfile.txt
```

3.4.3 Removing a file; rm file

The command `rm` *filename* (remove file) will delete a file. This is dangerous because you can't get it back. If you want the `rm` command to give you a chance to *not* delete a file, use the `-i` (interactive) flag.

```
username@alpha ~ $ rm -i file.txt
rm: remove regular file 'file.txt'? [you enter y or n here]
```

You can actually remove many files: either list them explicitly (`rm` *file1 file2 file3* etc.) or use a wildcard: `rm -i *.txt` will delete every file ending in `.txt` but prompt you before deleting each file. Note: `rm *.txt` will delete all those files **without** prompting you. Use it only if you like living dangerously [7].

3.4.4 Removing a directory; rmdir; rm -r directory

The `rmdir` lets you delete a directory but only if it's empty (contains no files or subdirectories).

```
username@alpha ~ $ rmdir d
rmdir: failed to remove 'd': Directory not empty
```

If this is not dangerous enough for you, the `-r` option makes `rm` *directory* behave recursively: It will destroy a directory and everything within it. Use `rm -ri` *directory* if you want to recursively crawl through the directory and its contents, with "Are you sure?" prompts as you go[8].

If you want to delete the files in a directory but not the directory itself, use `rm` *directory*`/*`

3.5 Aliases and Startup Files

3.5.1 Aliases

Tired of typing `ls -F` every time? The `alias` command can save you some typing: `alias` *command*`="`*quoted string*`"` tells the shell that every time you enter *command* at the shell prompt, the shell should substitute the quoted string. For example,

```
username@alpha ~ $ alias lsf="ls -F"
username@alpha ~ $ lsf
d/  file.txt  mycopy.txt
```

The really cool thing is that the command can be an already-existing command. After the `alias ls` below, plain old `ls` automatically expands to `ls -F`:

```
username@alpha ~ $ alias ls="ls -F"
username@alpha ~ $ ls
d/  file.txt  mycopy.txt
```

Note: The substitution is not done recursively (so our `ls` doesn't behave like `ls -F -F -F` ...), and the substitution is only done on the command name, not on later parts of the line, so `mkdir ls` creates a directory named `ls`; it doesn't try to do `mkdir ls -F` (which makes `mkdir` unhappy and causes it to print an error message).

---

[7] As with `mv`, it's possible that your default configuration for `rm` makes it behave like `rm -i`. In that case, if you don't want prompts before deleting things, use `rm -f`.

[8] Or `rm -rf` directory if you don't want prompts. (Be afraid; be very afraid.)

Similarly, if you want `rm` and `mv` to always behave like `rm -i` and `mv -i`, you can use

```
username@alpha ~ $ alias rm="rm -i"
username@alpha ~ $ alias mv="mv -i"
```

You can use the `alias` command by itself to list all your aliases. Given the three alias commands we've entered (for `ls -F`, `mv -i`, and `rm -i`), our output could be something like

```
username@alpha ~ $ alias
alias ls='ls -F'
alias mv='mv -i'
alias rm='rm -i'
(Plus other aliases, depending on your setup)
```

To remove an alias, use `unalias` *command*, as in `unalias rm`

3.5.2 Starting up with .bashrc; appending to a text file with >>

If you get tired of entering the same `alias` commands every time you log in, there's a `.bashrc` file you can add them to; when you start up a shell, the commands will be automatically executed. (Recall that the "`rc`" in `.bashrc` stands for "resource" and the leading dot makes the file hidden from `ls` commands that don't include `-a`.)

If you haven't read the section below on Text Editors, then opening up `.bashrc` and adding some lines to it might be nontrivial.[9] If you want to add some standard `alias` commands to `.bashrc` without learning a text editor first, you can actually use the `cat` command to **append** what you type to a file.

Recall `cat >` *filename* lets you type in text (separating lines with carriage returns) until you enter ctl-D to end standard input.[10] If you use ">>" instead of ">" for redirection, your text will be **appended** to the end of the file. So an easy way to add some `alias` commands to `.bashrc` is with the following:

```
username@alpha ~ $ cd
username@alpha ~ $ cat >> .bashrc
alias gcc='gcc -Wall -std=c99 -lm'
alias ls='ls -F'
alias mv='mv -i'
alias rm='rm -i'
^D
username@alpha ~ $ cat .bashrc
... (some other output) ...
alias gcc='gcc -Wall -std=c99 -lm'
alias ls='ls -F'
alias mv='mv -i'
alias rm='rm -i'
username@alpha ~ $
```

---

[9] There's a "`nano`" text editor that's very simple to use: `nano .bashrc` will open the file; you can maneuver using arrow keys and enter text by typing it. Ending the session with ctl-X will give you a prompt about saving the changes; say "y" if you like them, "n" if you don't.

[10] Recall that the *^D* (i.e., ctl-D) you enter won't be echoed to the screen.

(Note I snuck in an `alias` command that ensures that `gcc` (the GNU C compiler) always runs with the flags `-Wall -std=c99 -lm`; see the section on Compiling and Running C programs below.)

3.6 More unix?

This ends the discussion of basic unix — you should now know enough to get yourself into trouble. There are, of course, many more parts to unix; you might even want to learn some of them sometime.

The `man` (for "**manual**") command will give you instructions for using commands: `man ls` will give you all the other options for `ls`, `man rm` describes the `rm` command, `man bash` gives you the manual for the `bash` shell, and so on. Don't expect to understand or remember all the various features of the commands. (The `bash` manual is over 5500 lines long, for example.)

There are also manual pages things other than commands. For example, `man ascii` will give you the ASCII table, and `man printf` will give you the programming instructions for the C function `printf`.

Other things you'll probably want to learn about are

- More of the options for `ls`, `cd`, `cp`, `mv`, `rm`, `cat`, `man` ("`man man`" tells you about the man command)
- Using complex **relative pathnames** like `directory/subdirectory/subsubdirectory` or `../../uncle/first_cousin/first_cousin_once_removed/`
- Using *~username* as the home directory for a give user.
- **Shell variables**, like `$PATH`, and **shell commands** for looping, making decisions, etc.
- **Redirection** of standard input using "<"
- "**Piping**" the output of one command to the input of another using "|" (vertical bar)
- Running commands in the **background vs foreground**
- Using `-` (just dash) as the name for standard input or output, depending on the context
- **Various commands** like
  - `chmod` (change mode) to modify file permissions
  - `diff` to show you the differences between two text files
  - `du` (disk usage) to show how much disk space is in use
  - `grep` (get regular expression) and `egrep` (extended grep) to look for patterns of text
  - `id` to print out your user and group id numbers
  - `more` to display a file on the screen, page-by-page
  - `ps` (process status) to tell you what you're running
  - `sort` to sort files
  - `wc` (word count) to count the number of words or lines in a text file

3.7 Text Editors

Simple editors like `Notepad` on Windows are fine for making small lists and reading the `README` files that come with software, but they aren't great for writing programs. Complex word processor programs like Word or Pages aren't designed for writing programs either. If you want to write programs, you really need a text editor. The choice of editor is up to you, but it's

important. There are three editors available on `alpha` that are fairly popular. I'll list them in the order that I like them.[11]

- `vim`, or "`VI Improved`", is my current favorite. Based on `vi` (visual editor), it is programmable and has an extensive library. It emphasizes being able to navigate files quickly, and it is especially well suited for people who know how to touch-type. Its syntax highlighting abilities are unmatched. Use the `vim` command to get started. The first page has a tutorial, or just run the command `vimtutor`.

  Advantages: Special modes for certain kinds of programming; highly customizable and programmable, modal interface, a huge library of customizations written for it, tends to start (a lot) faster than `emacs`.

  Disadvantages: Steep learning curve, you have to use the escape key a lot to switch between command and insert mode, and it will become a religion and you will be compelled to fight for its honor when compared with `emacs`.

- `emacs`, started in 1976 (!) and has continued to grow and evolve. There are two popular variants, FSF emacs (from the Free Software Foundation) and `xemacs` (developed in part at the University of Illinois at Urbana). The FSF version is called `emacs` and the other `xemacs`. Most users will not notice a difference between the two, other than that `xemacs` uses a bit more graphics.

  You can start it from your shell; pass it the `-nw` option to tell it not to open a graphic interface. MS-Windows cannot understand the graphic X protocol used by unix.

  The first page will tell you how to start the tutorial.

  Advantages: special modes for certain kinds of programming; highly customizable and programmable, simple interface, a huge library of customizations written for it.

  Disadvantages: Steep learning curve, you have to use a lot of control key sequences to do things, and it will become a religion and you will be compelled to fight for its honor when it is compared with `vim`.

- `joe`, `nano`, etc. These are very simple editors, but they do the job. They are for the lazy programmer who is content second-rate tools and practices, but still doesn't want to bear the shame of programming with notepad.

  Advantages: very small, short learning curve.

  Disadvantages: Not programmable, very little power, not as readily available as `emacs` and `vim`, and nobody cares enough about them to fight for their honor.

3.8 File Transfers

If you're working on a file at home and want to copy it to some machine, you should use the SFTP protocol (Secure FTP = Secure File Transfer Protocol = FTP over SSH). A popular program for this is Filezilla (see https://filezilla-project.org/). Just download the client. If you're having trouble connecting, some people have found that specifically specifying port 22 helps, even though 22 is the default.

If you're running unix on your home machine and are really into command lines, there's also an `sftp` command you can use. First `cd` to whatever directory has the file, then type

```
username@homeMachine ~ $ sftp username@alpha.cs.iit.edu
Password:
```

[11] This section was written by Dr. Beckman with just minor changes by Dr. Sasaki

```
Connected to alpha.cs.iit.edu
sftp> put filename                  ← if you want to upload a file
Uploading ... filename ...
sftp> get filename2                 ← if you want to download a file
Fetching ... to filename2 ...
sftp> exit
username@homeMachine ~ $
```

As with all things unix, there are many commands and options for `sftp`; see your friendly `man` page for help.

4 Compiling and Running C Programs

4.1 Compiling Using gcc

We'll be using the GNU[12] C compiler, `gcc`. Specifically, to test your programs, we'll compile them on `alpha` using

`gcc -Wall -std=c99 -lm` *yourfile.c*

- The `-Wall` flag says to display all warnings. These can include warnings about fairly harmless things people sometimes do, warnings about things that might cause problems if you're not careful, and warnings about things that will cause runtime errors or incorrect results. (You know — "bugs".) It's very tempting to test your program after it compiles without `-Wall`, but honestly, most of those warnings indicate things you should fix.
- The `-std=c99` flag says to use the C99 standard version of C. (We all need to use the same version of C, otherwise chaos will ensue.)
- The `-lm` flag says to include the math library (what you want when you `#include <math.h>` in your program, so that you can access functions like `sqrt`).

It's good to include `alias gcc='gcc -Wall -std=c99 -lm'` in your `.bashrc` file (see section 3.5.2 above). That way, you only need to type `gcc` *yourfile.c* because the flags will be included automatically.

If you've broken up your program into multiple `*.c` files (probably not necessary for this course), then specify all of them on the same compile line; for example, `gcc` *file1.c file2.c* etc.

By default, the executable file produced by the compiler is called `a.out` and it is in the current directory. If you want the executable to have a different name, use `-o` (letter oh) to specify the name you want: `gcc` *yourfile.c* `-o` *executableFilename*

4.2 Executing Your Program

To run your program, you will probably have to run `./a.out` instead of just `a.out`

```
username@alpha: ~ $ ./a.out
```

This is because the default doesn't tell the shell to look in the current directory for programs. If you want the shell to look in the current directory (which may vary over time), enter

```
username@alpha: ~ $ PATH=$PATH:.
```

[12] GNU stands for "GNU's Not Unix". By "Unix," they mean the proprietary version of the 1980's. By "GNU's Not Unix", they refer to how GNU consists of free software.

This tells the shell to add dot to the list of directories searched when you run a program.

```
username@alpha: ~ $ a.out
```

In an ideal world, your program runs correctly the first time. If it doesn't, there are three general classes of symptoms:

- It prints the wrong output. This is the easiest case. Investigate the problem and fix your bugs (see below).
- It goes into an infinite loop. Press ctl-C to stop your program, then investigate and fix your bugs.
- It halts with a runtime error: *Segmentation fault* is popular — it's caused by using an illegal array index or bad pointer value. Explanation: The memory you get to use is chunked up into one or more "segments"; if you try to access a memory address outside your segment / segments, you get a segmentation error.

Investigating Your Bugs

In real life, you'll learn how to use development environments with fancy debugging tools. Our programs are simple enough that you should be able to debug them by adding extra print statements to figure out where it goes and what values variables have at different points in time.

One big warning: C doesn't do runtime array index checking, nor does it check pointers to make sure they contain correct memory addresses. This means that buggy array and pointer operations can use or update the wrong chunk of memory. When this happens, you may find variables changing value for no apparent reason, which can cause mystifying wrong output or apparently-inexplicable infinite loops. If `gcc` warns about using null or uninitialized pointers or about having a pointer to the wrong type of value, **heed its warnings and fix your code**.