Using Your Linux Account

Original text by Mattox Beckman, 2014-01-27
Modified by Jim Sasaki, 2019-03-08

Table of Contents

1 Introduction

Welcome!

As a CS student, you get an account on one of the Linux machines `fusion1.cs.iit.edu` or `fusion2.cs.iit.edu`[1]. Logging into `fusion1` will let you access `linux1.cs.iit.edu`, which is where your actual program-writing and -running should be done.

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. You don't have to use `linux1` when developing your program; 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 more-or-less the same way on your machine as on `linux1`. (Details like the exact path to your home directory or what shell prompts you see may differ. Also, you may need to upload a copy of the GNU C compiler, "`gcc`", however.) If your

[1] In this document, I'll just say "`fusion1`" instead of "`fusion1 or 2`"

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 `linux1` 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 `linux1` so long as you avoid non-standard libraries like `conio.h` (console IO on Windows). Remember, it's your responsibility to make sure your program runs on `linux1`.

2 First Login

2.1 Secure Shell

To log on, you need to use `ssh` (*secure shell*[2]"), which uses the *secure shell protocol SSH* to improve safety. Using an SSH client, you can remotely log in to `fusion1`, which will let you access `linux1`, which is where you should edit and run your programs. On Linux or Mac OS, bring up a terminal window; `ssh` should be available on the command line. On Windows, you may need to install the SSH client PuTTY (see `http://www.putty.org/`). Set it up to connect to `fusion1.cs.iit.edu` using SSH. If you need help with this, feel free to ask your TA about it.

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 `fusion1` and not some machine pretending to be `fusion1`, go ahead and accept the host key.[3] It will be stored and you won't see this message in future connections.

2.2 Initial Account Information

To log in for the first time, see `https://fusion.cs.iit.edu/` → `Docs`. You should already have an account on `fusion1`; if you don't, contact the `fusion` support group (see `https://fusion.cs.iit.edu/` → `Contact`).

Your initial password should appear in the email from `fusion1` that told you about your account. You'll need to replace it immediately after the first login. **Note**: In the initial password, careful to distinguish capital O from digit 0, and capital I from digit 1 from lower L. Copying your password from the email and pasting it into the terminal emulator is safest. Also, don’t worry if the cursor doesn’t move as you type in your password, — it’s just a safety feature.

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.

---

[2] A shell is a command-line-based interface to a unix-like operating system like Linux or Mac OS.

[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 shop, be circumspect.

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 `linux1` is called `bash`. In the previous section you saw how to log on. At the bottom of the screen you saw something like this:

```
-bash~4.1$
```

This line is called the **prompt**. Prompts are highly customizable; the one you see here says you're running version 4.1 of the bash shell. The exact details of what your prompt looks like may vary from the one above. In these notes, we'll just use "`bash $`" as a generic prompt.

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 `linux1`!

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`*.)

```
bash $ echo hello world!
hello world!
bash $
```

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.)

```
bash $ echo hello
hello
bash $ world!
world!: Command not found
bash $ echo hello
hello
bash $ 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.

```
bash $ echo hello world! > hello.txt
bash $
```

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.

```
bash $ echo hello world! > hello.txt
bash $ echo hello.txt
hello.txt
bash $ cat hello.txt
hello world!
bash $
```

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 `^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.)

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

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:

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

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 `linux1`, the result of your `ls` may vary.)

```
bash $ pwd
/home/students/all/username
bash $ 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.

```
bash $ mkdir mydir
bash $ 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.

```
bash $ mv hello.txt mydir
bash $ ls
mydir myfile.txt
bash $ ls mydir
hello.txt
```

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

```
bash $ cat hello.txt
cat: hello.txt: No such file or directory
bash $ 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

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

Some bash prompts include the path to the current directory as part of the prompt. If the path is "~" (a tilde), this is unix shorthand for the name of your **home directory**, the main directory into which your files are placed. A path like `~/mydir` indicates the `mydir` subdirectory of your home directory. If your prompt does include your path, you might see prompts like

```
bash ~ $          [if you're in your home directory]
bash ~/mydir $    [if you're in the mydir subdirectory of your home]
```

but again, this depends on how your bash shell has been set up for you.

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`

```
bash $ ls
hello.txt mydir   myfile.txt
bash $ 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. The `.` (dot) and `..` (dot dot) names indicate the current and parent directories, respectively.

```
bash $ ls -a
.     ..  .bashrc  .bash_profile    hello.txt    mydir    myfile.txt
```

(Your exact output may vary from the output above. There's also a `-A` flag that's like `-a` but omits . and .. from the listing.)

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`:

```
bash $ ls -aF
./    ../ .bashrc  .bash_profile    hello.txt    mydir/   myfile.txt
```

The `ls -a` outputs above include are four hidden files.

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.

```
bash $ 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:

```
bash $ 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.

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

3.3.4 Wildcards

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

```
bash $ ls
mydir  myfile.txt
bash $ ls *.txt
myfile.txt
bash $
```

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).

```
bash $ ls -F
mydir/  myfile.txt
bash $ ls *
mydir/  myfile.txt
```

---

[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.

```
mydir:
hello.txt
bash $
```

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*

```
bash $ ls mydir
hello.txt
bash $ cp myfile.txt mydir
bash $ ls mydir
hello.txt myfile.txt
bash $ cp myfile.txt mycopy.txt
bash $ 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*

```
bash $ mv myfile.txt file.txt
bash $ mv mydir d
bash $ 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:

```
bash $ ls
d     file.txt    mycopy.txt
bash $ 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.

---

[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.

```
bash $ mv d file.txt
mv: cannot overwrite non-directory 'file.txt' with directory 'd'
```

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

```
bash $ mkdir d2
bash $ mv d2 d
bash $ 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.

```
bash $ 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).

```
bash $ 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,

```
bash $ alias lsf="ls -F"
bash $ lsf
d/  file.txt  mycopy.txt
```

---

[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 **really** don't want prompts. (Question: What does "`rm -rf .`" do? Be afraid; be very afraid.)

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

```
bash $ alias ls="ls -F"
bash $ 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).

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

```
bash $ alias rm="rm -i"
bash $ 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

```
bash $ 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 `^D` (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:

```
bash $ cd
bash $ cat >> .bashrc
```

[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 `^X` (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` (ctl-D) you enter won't be echoed to the screen.

```
alias gcc='gcc -Wall -std=c99 -lm'
alias ls='ls -F'
alias mv='mv -i'
alias rm='rm -i'
^D
bash $ cat .bashrc
... (some other output) ...
alias gcc='gcc -Wall -std=c99 -lm'
alias ls='ls -F'
alias mv='mv -i'
alias rm='rm -i'
bash $
```

(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 `linux1` that are fairly popular. I'll list them in the order that I (Dr. Beckman) like them.

- `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 `scp` (secure copy). See `https://fusion.cs.iit.edu/` → `Docs` for details.

4 Compiling and Running C Programs

4.1 Compiling Using gcc

We'll be using the GNU[11] C compiler, `gcc`. Specifically, to test your programs, we'll compile them on `linux1` 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 ignore the warnings from `-Wall` (or to compile your program 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@linux1: ~ $ ./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@linux1: ~ $ PATH=$PATH:.
```

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

```
username@linux1: ~ $ a.out
```

[11] 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.

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 `^C` (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.

4.3 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 truly mystifyingly, seemingly inexplicably bad output or 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**.