# Table Of Contents



## Getting onto the Zoo

Type "`ssh <netid>@node.zoo.cs.yale.edu`", and enter your netID password when prompted. Your password won't show up and the cursor won't move, but it knows what you're typing.

## Getting Files onto the Zoo

Use the "`scp`" command to transfer files onto the Zoo. Type "`scp filename1 filename2 <netid>@node.zoo.cs.yale.edu:~/mydirectory/mysubdirectory`"
You should be on your local machine in the same directory as the two files. This will copy the two local files, `filename1` and `filename2` into the directory `mysubdirectory` (which is located inside `mydirectory` in your home directory on the Zoo). You will be prompted to enter your netID password, just like when you ssh in.

## Unix Basics

*What is a directory?* A directory is simply a folder, which can contain any mixture of files and subdirectories.
*What is different about a file versus a directory?* A directory contains other files and directories, but does not contain information itself -- for example, you can't use emacs to view a directory. However, you can often use a text editor to view the contents of a file. Sometimes directories and files have different colors when you look at what's in your current directory, depending on your terminal settings, which can help you tell which is which.
*What about an executable?* An executable is a special type of file that can be run by your computer. In the case of examples from class, executables include "`Distance`" and "`Split`". To run an executable, type "./" before the name. For example, to run `Distance` without any input arguments, you could type "`./Distance`". With command-line arguments, it might look something like "`./Distance 52.0 0.0 42.0 -72.9`". Note that if you try viewing an executable with a text editor, it will look like nonsense! Executables are created by compiling your source files (for example, `distance.c`).

Essential Commands

**pwd** - print working directory. This command will tell you the exact file path of your current location, and requires no further arguments

**ls** - list the files in your current directory. For example:

```
-bash-4.3$ pwd
/c/cs223/hw1
-bash-4.3$ ls
final  Makefile  t1  t2  t5  test  Tests  Total  Total.c
Total.o  Totalx
```

*Special options*:

1. "ls -l" allows you to use a long listing format, which provides extra information (such as the file permissions, owner, date and time of last edit, etc.) Rows with start with d, such as the row corresponding to "`Tests`" below, are directories!

```
-bash-4.3$ pwd
/c/cs223/hw1
-bash-4.3$ ls -l
total 76
drwxrws--- 4 sbs5  cs223ta  4096 Feb  3 10:14 final
-rw-r--r-- 1 sbs5  cs223ta  5582 Jan  3 12:47 Makefile
-rw-rw-r-- 1 sbs5  cs223ta    35 Jan  3 12:47 t1
-rw-rw-r-- 1 sbs5  cs223ta    44 Jan  3 12:47 t2
-rw-rw-r-- 1 sbs5  cs223ta    11 Jan 23 14:09 t5
-rwxrwxr-x 1 sg686 cs223ta  8872 Jan 22 19:37 test
drwxrwsr-x 2 sg686 cs223ta  4096 Feb  2 20:23 Tests
-rwxrwxr-x 1 sbs5  cs223ta  8872 Jan 25 16:05 Total
-rw-rw-r-- 1 sbs5  cs223ta   866 Jan 25 16:05 Total.c
-rw-rw-r-- 1 sbs5  cs223ta  2112 Jan 25 16:05 Total.o
-rwxrwxr-x 1 sbs5  cs223ta 13208 Jan 22 14:51 Totalx
```

**cd -** cd, or "change directory", allows you to move from one directory to another. For example, if you are on the zoo and want to get to the cs223 folder, you could type: "`cd /c/cs223`"

*Special options*:

1. "`cd`", with no extra arguments, will bring you to your home directory: on the zoo, this would be "`/home/accts/<netid>`"
2. "`cd ..`", with precisely two periods, will move you "up" one directory. See the following example:

```
-bash-4.3$ cd /c/cs223/hw1
-bash-4.3$ pwd
/c/cs223/hw1
-bash-4.3$ cd ..
-bash-4.3$ pwd
/c/cs223
```

**mv** - move a file from one location to another. You have a few options with mv:

1. "`mv origfile directory`" will move origfile from your current directory to directory. Note that if you wish to move a file that is not in your current directory, you can simply replace the file name with its complete file path (eg /homedir/subdirectory/origfile)
2. "`mv origfile newname`" simply renames origfile to newname. If there is already a file called newname, mv will clobber that file, replacing it with the contents of origfile.

**cp** - copy a file. There are a few options for cp, as shown below:

1. "`cp origfile newfile`" creates a new file called `newfile` in your current directory, with the same contents as `origfile`. As with mv, cp will overwrite existing files without warning.
2. "`cp origfile /directory/subdirectory`" makes a new copy of `origfile` (with the same name, `origfile`), in `/directory/subdirectory`. Note that if there is already a file called `origfile` in `/directory/subdirectory`, it will be overwritten!
3. "`cp origfile /directory/subdirectory/newname`" also makes a new copy of `origfile` in `/directory/subdirectory`, but calls it `newname` instead of `origfile`

**cat -** print the content of a file to standard output using "`cat <filename>`"

**rm** - remove a file using "`rm <filename>`". Be very careful with this command - you can't get a file back once you have deleted it!

- Use the "-r" option to recursively delete a directory: "`rm -r <directory>`". Again, be careful: this will delete the directory, all the files in the directory, and all subdirectories (and all the files in those subdirectories, etc.)

**echo** - on its own, echo simply outputs the argument it was passed to standard output. However, certain characters (" and \) will be treated irregularly. As such, if you wish to echo a string that contains either character, it is necessary to either enclose the string in double quotes, or precede each character by a backslash, \, so that bash will pass echo the desired phrase. See below:

```
-bash-4.3$ echo word
word
-bash-4.3$ echo "word"
word
-bash-4.3$ echo \1
1
-bash-4.3$ echo "\1"
\1
-bash-4.3$ echo \\1
\1
-bash-4.3$ echo \"
```

"

In fact, since this behavior is a feature of bash, you may find escape characters relevant outside of echo. For example, suppose you wish to create a file called my"file. If you try to just type touch my"file, it won't work. Instead, you must either escape the ", or enclose the entire name in quotes, as shown below:

1. This doesn't work:
   ```
   -bash-4.3$ touch my"file
   > ^C
   ```
2. Option 1:
   ```
   -bash-4.3$ touch my\"file
   ```
3. Option 2:
   ```
   -bash-4.3$ touch "my\"file"
   ```

*Special options:*

1. "`Echo -n`" prevents echo from "echoing" the trailing newline character. See below:
   ```
   -bash-4.3$ echo sampleword
   sampleword
   -bash-4.3$ echo -n sampleword
   sampleword-bash-4.3$
   ```
2. "Echo -e" tells echo to take whatever input it gets from bash, and interpret backslash escapes. This is a bit subtle, so here's an explanation of the following three commands.

(1) Bash turns `word\word` into `wordword`, so echo doesn't see any backslashes and the `-e` flag has no effect

```
-bash-4.3$ echo -e word\word
wordword
```

(2) Bash turns `word\\word` into `word\word`, but `\w` is not a known escape character, so the `-e` flag has no effect

```
-bash-4.3$ echo -e word\\word
word\word
```

(3) Bash turns `word\\nword` into `word\nword`, so echo receives `word\nword`. Then, `-e` notices the `\n`, which it recognizes as a newline character thanks to the `-e` flag. Compare this to the behavior with the `-e` flag, below!

```
-bash-4.3$ echo -e word\\nword
word
word
-bash-4.3$ echo word\\nword
word\nword
```

**< and >** - these two operators allow you to redirect standard input and output. (Recall that echo returns your input string to standard output, while Packx read from standard input.) < allows you to redirect standard input, while > allows you to redirect standard output. See examples below:

1. Standard output redirection: "`echo word1 word2 >filename`" creates a new file called filename, which says "`word1 word2`" (followed by a newline character). Note that if you want to use standard output as the input *to an executable*, you should not use

redirection -- instead, use the pipe command shown below! If there is already a file named "filename", output redirection will **overwrite** it with the standard input supplied
2. Standard input redirection: "cat <filename" feeds filename as the input to the cat command, listed above

**|** - known as the "pipe", this allows you to feed the standard output of the lefthand side argument, into the standard input of the righthand side argument. For example, you can use this with the distance calculator example: "`echo 52.1 0.0 42.5 -72.9 | ./Distance`"

Other Sources of Unix Information

This is far, far from everything Unix can do, and there are likely lots of other commands you might find that can save you some time as you navigate the Zoo. For Professor Slade's more extensive introduction to Unix, click **here**. For a short list of Unix commands which may be helpful, click **here**. Finally, if you ever have questions about what an option for a particular command does (for example, if you see "`ls -a`" and aren't sure what the "`-a`" option accomplishes), try the built-in manual page! To access this page for a particular command, use the man command, followed by the name of the Unix command: for example, you could type "`man ls`" and scroll through. Try it!

**Homework Testing**

1. You must be on the Zoo, and *in the same folder* as your completed homework files and executable. The test script will test the executable in *your current directory*, not whatever file you have submitted!
    a. To test this: use the `ls` command, and make sure you see your homework files: the main .c file (comments.c, no_ap.c), the Makefile, and the executable, which by local convention has the same name as the .c file without a file type and with the first letter capitalized. Sometimes the executable name is a different color than the other files, depending on your zoo settings.
    b. If you don't see an executable, that means you haven't compiled your program. To create the executable to run your program, use the command "`make`" or "`make <program>`" where <program> is the name of your .c file without the ".c" part. To understand how makefiles work, you can read this or the assigned reading from the Linux Programming book.
    c. If you updated your .c file, you have to run the "make" command again to update the executable, so the changes will appear when you run the test cases.
2. To run all the tests at once, type: "`/c/cs223/hw<#>/Tests/test.<Name>`" If your code fails a test, the output of the test script will include the result of running `diff` on the expected output and your program's output. See StackOverflow for an explanation of `diff` output, or examine the input and output files as described below.
3. To run one test at a time, for hw1: "`./Comments < /c/cs223/hw1/Tests/t01.c`"
    a. This format was necessary for hw1 because your program had to read from standard input. However, for hw2 (and other programs which depend on command line arguments), you can simply run the executable t01: "`/c/cs223/hw2/Tests/t01`"

4. The expected output for each individual test can be found in the .cs files, so to view the expected output of a test you could type "`cat /c/cs223/hw1/Tests/t01.cs`". You could also use your favorite text editor, eg "`emacs /c/cs223/hw1/Tests/t01.cs`"
    a. For homework 2, the `.cs` extension has been replaced by the `.p` extension. So, if you wanted to see the expected output of test 3 of homework 2, you could type "`cat /c/cs223/hw2/Tests/t03.p`"
    b. Remember that you can use the "`ls /c/cs223/hw1/Tests/`" command to see all the files related to testing

**Homework Submission**

1. To submit your homework, you need to be on the Zoo, and in the same folder as your completed homework files
    a. To test this: use the `ls` command, and make sure you see your homework files
2. Type: "`/c/cs223/bin/submit <homework #> Makefile <.c / .h homework files> <logfile>`"
    a. Example, for homework 1:
    `/c/cs223/bin/submit 1 Makefile Total.c time.log`