Unix In Depth

Control Characters

```
ctl-m           //return
ctl-d           //logout - same as typing exit
ctl-g           //rings bell on the terminal
ctl-h           //backspace
ctl-u           //delete whole line
ctl-s           //pauses output to screen
ctl-q           //un-pauses output to screen
```

Unix Mail

```
mail            //to read your mail
Return          //advances to the next email
d               //deletes the email
p               //reprint the email
s <filename>    //saves it in a file that you name
q               //quit mail
mail <user>     //opens a new email to <user>
ctl-d           //sends the email and closes the mail program
```

LS Options

```
ls -t               //sort by time
ls -u               //gives info on when files were used
ls -r               //reverses order of any other option used (e.g
. -t)
ls -d               //check just the directory that you're in
ls -c               //
ls -i               //reports the i-number of each file (decimal n
otation)
```

CMP

```
cmp                 //compares 2 files byte by byte - diff better
```

`cmp` works on any type of file, although `diff` only works on text files.

Directories

```
pwd                 //print working directory
echo *              //echoes all the non-hidden files in the direc
tory
cat *               //prints all the files in the dir
rm *                //deletes all files in the current directory
```

Files

```
file                //determines what type a file is
```

A runnable program is maked by a binary 'magic number' at it's beginning. Use `od` with no options to find it. The octal value `410` marks a purely executable program. `410` is not ASCII text, so an editor cannot create it.

In Unix there is only one type of file, and all that is required to access it is its name.

Misc

```
&                //if you end a command with &, it will start t
he                     //program but accept further prompts
od               //octal dump. Shows the bytes of a file. Use w
ith -cx
```

Programs retrieve the data in a file by a system call (a subroutine in the kernel) called `read`. Each time `read` is called, it retrieves the next part of a file. E.g. the next line of text typed on the terminal.

```
rm -f            //forces removal without interactive request
```

Parentheses can be used to group commands. Here the output of `date` and `who` are concatenated into a single stream that can be sent down a pipe.

```
(date; who) | wc
```

You can grab the interum output from a command that is being piped and store it in a file with the `tee` command. E.g.

```
(date; who) | tee save | wc
```

WC receives the data as if `tee` weren't int he pipeline.

```
test
```

tests if a file exists. returns an exit status. The shell stores the exit status of the last program in `$?`.

Processes

An instance of a running program is called a process. Processes are not the same as programs.

Every time you run the **program** `wc`, it creates a new **process**.

If several instances of the same program are running at the same time, each is a separate process with a difference process ID.

```
kill 0
```

This will kill all your processes besides the login shell

```
nohup <command> &
```

The command will continue to run if you log out and will save any output into the file `nohup.out`

```
nice <resource heavy command> &
```

If you have a command that uses up a lot of processor resources, you can run it with lower priority, so that other users don't suffer.

```
at time
<commands>
ctl-d
```

This will run a command at whatever time you like.

SHELL Variables

```
$PS1            //Terminal Prompt
$PATH           //Search Path
$TERM           //Name of Terminal you're using
```

To tell other programs that you want to use a personal variable you've set in `.bash_profile` or just in the terminal, use `export`. e.g.

```
export d="/dev/"
```

Permissions

When you login, you are assigned a `uid` by the system. 2 different login names can have the same `uid`, making them indistinguishable to the system, although this is not good for security reasons.

Every new user is assigned to the group of `Other`, although this varies by system.

In `/etc/passwd` you'll find all the passwords for all users of the system. While the file is ordinary text, the field definiteions and separators are agreen upon conventions used by the programs that use the file.

```
login-in:encrypted-password:uid:group-id:mescellany:login-dire
coty:shell
```

So for my Unix configuration, it's

```
root:x:0:0:root:/root:/bin/zsh
```

If the shell field is empty, it implies that you use the default shell. The miscellany filed may comtain anything (phone number/postal address).

When you give you password to `login`, it encrypts it and compares the result against the encrypted password in `/etc/passwd`. If they agree, it logs you in.

The file `/etc/group` encodes group names and group id's, and defines which users are in which groups. `/etc/passwd` only identifies users in your login group.

```
newgrp              //changes your group permissions to anothe
r group                       //and logs you into that group
```

To change your password, use the `passwd` command.

If you use `which passwd` to find it's path, then use `ls -lah /usr/bin | grep passwd` (or whatever it's path is), you'll see `passwd` 's permissions.

Note that instead of `-rwxr-x-r-x` , it has `-rwsr-xr-x` . The `s` in the execute field means that when the command is run, it is to be given the permissions corresponding to the file owner (i.e. root). This means that any user can run the `passwd` command to edit the password file.

What **executable** means: when you type something like `who` to the shell, it looks in a set of directories name `who` . If it finds the file, and has the execute permission, the shell calls the kernel to run it. The kernel checks the permissions, and if valid, runs the programm.

NOTE: A program is just a file with excecute permissions.

If you have write permission to a directory, you can delete files in that directory, even if you don't have write permissions to those files.

If you `chmod` a file or directory, it won't update its modification date. That only happens when you modify the contents of a file/dir.

Inodes

Administrative information, such as permissions, modification dates, disc location, and file size are not stored in the file itself, but in a system structure called an index node, or **Inode**.

There are 3 times in the Inode - last modified, last used (read or executed), last change of Inode itself.

The system's internal name for a file is its **i-number** - the number of the Inode holding the file's information.

The i-number is stored in the first 2 bytes of a directory, before the name. `od -d` will show you this. These 2 bytes are the only connection between the filename and its contents. Therefore a filename in a dir is actually a *link*, because it links the name in the directory hirarchy to the Inode, and hence the data.

The same i-number can appear in more than 1 directory. The `rm` command removes links, and when the last link to a file disappears, the system removes the Inode itself, and hence the file.

The number printed between permissions and owner with the `ls -lah` command is the number of links to the file. There is no difference between the first link and subsequent ones.

Devices

Instead of system routines to control devices, there are files in `/dev` that contains device information that the kernel references before issueing hardware commands.

If you do `ls -l /dev`, the first char of the permissions will be either `b` or `c`. For a device file, the inode contains the internal name for the device, which consists of its type - character `c` or block `b`, and a pair of numbers calle dthe major and minor device numbers.

The major number encodes the type of device, while the minor number distinguishes different instances of the device.

Disks are block devices, and everything else is a character device. On Mac Unix, nearly everything is a `c`. Only 4 or so `b`'s

`mesg n` Will turn off messages `mesg y` Will turn them back on again

To time a command without your screen getting filled up with junk output, you can use `/dev/null`. E.g.

```
time ls -R / > /dev/null
```

You get

```
real	0m21.931s
user	0m2.174s
sys	0m3.378s
```

In order, these times are elapsed clock time, CPU time spent in the program, and CPU time spent in the kernel while the program was running.

The Shell

Metacharacters

Characters like `*` that have special properties are known as metacharacters. There are a lot of them. To stop a character from being interpreted as a metacharacter, enclose it in single quotes `'`.

Double quotes don't work as well, because the Shell still looks inside for a `$` or a `\`.

Another way is to escape every instance of the metacharacter with a slash `\`.

A `\` at the end of the line tells the shell to ignore the line break.

Creating New Commands

One way to create a new command is to create a file that contains the set of commands you want to execute. E.g.

```
echo 'who | wc -l' > nu
```

Then you can call it from the shell in 2 ways:

```
sh < nu
sh nu
```

If a file is executable and contains text, then the shell assumes that it is a file of shell commands. That said, you still have to place it in one of the directories in $PATH, or add the current directory to $PATH.

```
chmod +x nu
```

Now you can run the command just by typing `nu`.

Command Arguments and Parameters

When the shell executes a file of commands, each occurance of $1 is replaced by the first argument, each $2 is replaced by the second argument, and so on until $9.

Eg. if you make

```
chmod +x $1
```

with the command

```
cx nu
```

it will take `nu` as the first argument.

To make a command take an unlimited number of arguments, use `$*`. E.g.

```
chmod +x $*
```

Although if an argument with more than 1 word is supplied (e.g. "elis island"), it will throw an error, even if the argument is in quotes, because bash will interpret the space as a delimiter between 2 difference arguments, so

```
echo 'grep $* phone.txt' > grop
echo 'elis island' > phone.txt
chmod +x grop
grop "elis island"
```

Will throw an error. What you need to do is encase the `$*` in double quotes, since bash will look inside it for instances of `$`, `\`, and `...`

```
echo 'grep "$*" phone.txt' > grop
```

Now arguments with a space will work.

The argument `$0` is the name of the program being executed. So in the example above, it would be `grop`.

Commands in the Sub-Shell

Commands are carried out in a sub-shell. This means that without modification, they cannot set shell variables, because variables are associated with the shell they are created in, and are not inherited by child shells. Unix provides a dot operator `.` that executes commands in the current shell, rather than a sub-shell. Unfortunately, it can't be used in files full of commands - tested on mac.

You can't pass arguments to a command prefixed by `.` so you can't use `$1` `$2`, etc

If you set shell variables in a sub-shell, they are only available in the sub-shell, until you declare them using the `.` operator.

When you want to make the variable available in sub-shells, you should use the `export` command.

Although, if you export a shell variable in a sub-shell, it will not be avaiable in the parent shell.

Echoing Commands

Using batsticks (`), you can echo out commands:

```
echo hello `date`
```

Command Exit Status

Every command returns an **exit status** – an integer value returned to the shell to say what happened.

```
0        //true - the command ran successfully
not 0    //false – the command did not run successfully
```

Since there are many different options for false, they are sometimes assigned reasons for failure. E.g. with `grep` , 0 = match, 1 = no match, 2 = error in pattern or filenames.

The command `exit` can be used to return an exit status.

Redirecting the Standard Error Output

Every program has 3 default files that are created when it starts, numbered by small integers called file descriptors.

The standard input is `0` . The standard output is `1` which is often redirected from and piped into. The standard error output is `2` .

Sometimes programs produce output on the standard error even when they work properly, e.g. `time wc` .

```
time wc desktop 2>tmr.txt
```

This will store the standard error output in `tmr.txt` . It also works for any error message, e.g.

```
fdsaf 2>error.txt
```

No spaces are allowed between `2` and `>` and `filename`

The notation `2>&1` tells the shell to put the standard error on the same stream as the standard output. The notation `1>&2` tells the shell to put the standard output on the same stream as the standard error.

It can help stop output disappearing into a pipe or other file. Very useful.

The shell allows you to put the standard input for a command along with the command, rather than a separate file, so the shell file can be completely self-contained.

There's something called a `here document`, which takes standard input from the first delimiter until the next delimiter, then substitutes for `$`, `...`, and `\`.

```
<<s          //regular here document
<<\s         //no substitution
<<'s'        //no substitution
```

Looping in the Shell

The shell is a programming language with variables, loops, and decision making capabilities.

A `for` statement reads as follows:

```
for var in list of words
do
    commands
done
```

e.g.

```
for i in *
do
    echo $i
done
```

The `i` can be any shell variable, although `i` is traditional. Note that the var's value is accessed by `$i` but the `for` loop refers to the var as `i`. `*` is used to pick up all the files in the `pwd`, but any other list of args can be used.

You can also write a `for` loop as such:

```
for i in <list>; do <commands>; done
```

You should use the for loop for multiple commands, or where the built-in argument processing in individual commands is not suitable.

Usually, the argument list in a for loop comes from file names, but it can come from anything. E.g.

```
for i in `cat list.txt`
```

Or you can just type in the arguments regularly. E.g.

```
for i in 3 4 5 6; do ln 2 $i; done
```

Filters

Filters are programs that read some input, perform a simple transformation on it, and write some output. E.g. `grep`, `tail`, `sort`, `wc`, etc.

grep

```
grep -n var *.[ch]      //locate var in C source
grep From $MAIL         //print message headers in mailbox
grep From $MAIL | grep -v mary      //headers that didn't come
 from mary
grep -y mary $HOME/lib/phone        //find mary's phone number
who | grep mary         //see if mary is logged in
ls | grep -v temp       //filenames that don't contain temp

-n      //prints line numbers
-v      //inverts the test
-y      //lower case match upper case, but upper case don't ma
tch lower case
```

`egrep` accepts extended regular expressions, using parentheses `()`, OR operators `|`, and the `+` and `?` operators.

`fgrep` was created because it can handle pattern sizes much larger than `grep` and `egrep` because it looks for literal strings only - i.e. no metacharacters like `*` and `?`.

It can efficiently look for thousands of words in parallel.

SED

`sed` has a heap of stuff you can do with it. Look for Table 4.2: Summary of sed Commands on PDF page 125, Book page 113.

Shell Programming

The shell is really a programming language in which each statement runs a command.

A major theme in shell programming is making programs robust so they can handle improper input and give helpful information when things go wrong.

One common use of a shell program is to enhance or to modify the user interface to a program.

For example, if you type the wrong arguments into a command like `cal`, you'll get the wrong result. E.g.

```
cal 10          //won't give you the month of October
```

No matter what interface the `cal` command provides, you can change it without changing `cal` itself. You can place a command in your private `bin` directory that converts a more conventient argument syntax into whatever the real `cal` requires. You can even call your version `cal`, which means one less thing for you to remember.

You need to decide how man arguments there are, then map them to what the standard `cal` wants.

The shell provides a `case` statement to help make such decisions.

```
case <word> in
pattern) commands ;;
pattern) commands ;;
…
esac
```

The `case` statement compares `<word>` to the patterns from top to bottom and then performs the commands associated with the first, and only the first, pattern that matches. Must be ended with `esac`

To the best of my knowledge, this works like a `switch` statement in Java or PHP.

The patterns are written using the shell's pattern matching rules, slightly different from the regular filename matching rules.

Our version of `cal` decides how many arguments are present, processes them, the calls the real `cal`. The shell variable `$#` holds the number of arguments that a shell file was called with.

Shell Built-in Variables

```
$#      //the number of arguments
$*      //all arguments to shell
$@      //similar to $*
$-      //options supplied to the shell
$?      //exit status of the last command executed
$$      //process-id of the shell
$!      //process-id of the last command started with &
$HOME   //default argument for cd command
$IFS    //list of characters that separate words in arguments
$MAIL   //file that, when changed, triggers "you have mail" me
ssage
$PATH   //list of directories to search for commands
$PS1    //prompt string, default '$ '
$PS2    //prompt string for continues command line, default '>
 '
```

What is the difference between `$@` and `$*`? * `$*` and `$@` expand into the arguments, and are rescanned; blanks in arguments will result in multiple arguments * `"$*"` is a single word composed of all the arguments to the shell file joined together with spaces. * `"$@"` is identical to the arguments received by the shell file; blanks in arguments are ignored, and the result is a list of words identical to the original arguments.

Shell Pattern Matching Rules

```
*       //match any string
?       //match any single character
[a-z]   //match any of the characters in the range of a-z
"..."   //match ... exactly; quotes protect special characters
. Also '...'
\c      //match c literally
a|b     //matches either a OR b
/       //in filenames, matched only by an explicit / in the e
xpression; in case, matched like any other character
.       //as the first characters of a filename, is matched on
ly by an explicit . in the expression
```

The SET Command

```
date
set `date`
echo $1; echo $4
```

`set` is a shell built-in command. With no arguments, it shows the values of variables in the environment. Ordinary arguments reset the values of `$1`, `$2`, etc. So

```
set `date`
```

sets `$1` to the day of the week, `$2` to the name of the month, etc.

`set` has several options. `-v` ad `-x`. They turn on the echoing of commands as they are being processed by the shell. These are great for debugging shell programs.

The IF Statement

The shell's `if` statement runs commands based on the exit status of a command. E.g.

```
if <command>
then
    commands if condition true
else
    commands if condition false
fi
```

`fi`, `then`, and `else` are only recognized after a newline or semicolon. The `else` is optional.

The `if` statement always runs a command – the condition – whereas the `case` statement does pattern matching directly in the shell.

`case` statements are much faster than `if` statements, so use `case` when you can.

A `case` statement can't easily determine permissions on a file. That is better done with a `test` and an `if`.

The OR and AND Statement

The shell provides two other operators for combining commands – `||` and `&&`.

```
test -f filename || echo file filename does not exist
```

is equivilent to

```
if test ~ -f filename
then
    echo file filename does not exist
fi
```

The command to the left of `||` is executed. If its exit status is `0` (success), the command to the right of `||` is ignored. The the left side returns non-zero (failure), the right side is executed and the value of the entire expression is the exit status of the right side.

The `&&` statement executes its right hand command only if the left one succeeds.

Loops

There are 3 loops: `for`, `while`, and `until`. The `for` loop is the most commonly used.

The `for` loop executes a set of commands - the loop body - once for each element of a set of words. Most often these are just filenames.

The `while` and `until` loops use the exit status from a command to control the execution of the commands in the body of the loop. The loop body is executed until the condition command returns a non-zero status (whore `while`) or zero (for `until`).

`while` and `until` are identical except for the required exit status to terminate the loop.

Here are the basic forms of each loop:

```
for i in <list of words>
do
    loop body, $i is set to successive elements of list
done

for i           #List is implicitly all arguments to the shell
 file, ie, $*
do
    loop body, $i set to successive arguments
done

while command
do
    loop body executed as long as command returns true
done

until command
do
    loop body executed as long as command returns false
done
```

`:` is a shell built-in command that only evaluates its arguments and returns "true".

The `true` command merely returns a "true" exit status. `false` returns a "false" exit status.

`:` is more efficient than `true` because it does not execute a command from the file system.

You can place a `break` in a `for` loop. This is borrowed from C, and has the effect of terminating the innermost enclosing loop. Sometimes, it will break out of the `for` loop when `q` is typed.

More on Shell Variables

`${var}` is equivalent to `$var`, and can be used to avoid problems with variables inside strings containing letters or numbers

```
var=hello
varx=goodbye
echo $varx       //goodbye
echo ${var}x     //hellox
```

Certain characters inside the braces specify special processing of the variable. If the variable is undefined, and the name is followed by a question mark, then the string after `?` is printed and the shell exits (unless interactive). If the message is not provided, a standard one is printed.

```
echo ${var?}        //"hello" — var is set
echo ${junk?}       //"junk: parameter not set" — default mess
age
echo ${junk?error!}     //"junk: error!" — Message provided
```

Note that the message generated by the shell always contains the name of the undefined variable.

Another form is ${var-thing} which evaluates to `$var` if it is defined, and `thing` if it is not. `${var=thing}` is similar, but also sets $var to `thing`:

```
echo ${junk-'Hi there'}          //"Hi there"
echo ${junk?}                    //"junk: parameter not set" —
junk unaffected
echo ${junk='Hi there'}          //"Hi there"
echo ${junk?}                    //"Hi there" — junk set to Hi
there
```

So `t=${1-60}` sets `t` to `$1`, or if no argument is provided, to `60`.

Evaluation of Shell Variables

```
$var         //value of var, nothing if var undefined
${var}       //same; useful if alphanumerics follow variable na
me
${var-thing}    //value of var if defined, otherwise thing. $v
ar unchanged
${var?msg}      //if defined, $var. Otherwise, pring msg and e
xit shell. If                     //message empty, print var:
parameter not set
${var+thing}    //thing if $var defined, otherwise nothing
```

TRAP – Catching Interupts

If you hit `Ctl + c` while a program is running, it can leave temporary files around. You need to detect if this happens and catch it.

When you hit `Ctl + c`, an interupt signal is sent to all the processes that are running of the terminal. When you hang up, a hangup signal is sent . There are other signals as well.

Unless a program has taken explicit steps to deal with signals, the signal will terminate it.

The built-in shell command `trap` sets up a sequence of commands to be executed when a signal occurs:

```
trap sequence-of-commands list of signal numbers
```

The sequence-of-commands is a single argument, so it must be almost always be quoted. The sigal numbers are small integers that identify the signal.

```
0       //shell exit (for any reason, including end of file)
1       //signal generated by hanging up
2       //signal generated by pressing `Ctl + c`
3       //quit (Ctl+\; causes program to produce core dump)
9       //kill (cannot be caught or ignored)
15      //terminate, default signal generated by kill
```

To put a `trap` before a loop in case anything goes wrong:

```
trap `rm -f $new $old; exit 1' 1 2 15
while :
```

The command sequence that is the first argument of `trap` - `rm -f $new $old; exit 1` is executed immediately if the signal happens. When it finishes, the program that as running will resume where it was unless the signal killed it.

Therefore, the `trap` command sequence must explicitly invoke `exit` , otherwise the other program will continue to execute after the interrupt.

Also, the command seuqnce will be read twice: once when `trap` is set and once when it is invoked. Therefore, the command sequence is best protected with single quotes, so variables are evaluated only when the trap routies are executed.

`trap` can help to prevent a program from being killed by the hangup signal generated by a broken phone connection:

```
(trap '' 1; long-running-command) &
```

The NULL command sequence means "ignore interupts" in this process and its children. The parentheses cause the `trap` and command to be run together in a background sub-shell. Without them, the `trap` would apply to the login shell as well as to long-running-command.

The `nohup` command is a short shell program to provide this service.

`test -t` tests whether the standard output is a terminal, to see if the output should be saved.

`exec` is a built-in shell command that replaces the process running in this shell by the named program, thereby saving one process – the shell that would normally wait for the program to complete.

The KILL Command

```
kill -9 process id
```

is how the signal 9 is sent. It can't be caught or ignored. It always kills.

`kill -9` is not the default because a process killed that way is given no chance to put its affairs in order before dying.

Sorting Properly

```
sort file1 -o file2
sort file1 > file2
```

There 2 are nearly the same. Using `>` will truncate the input file before it is sorted. The `-o` option, though, works correctly, because the input is sorted and saved in a temp file before the output file is created.

Shift

The shell builtin command `shift` moves the entire argument list one position to the left. `$2` becomes `$1`, `$3` becomes `$2`, etc.

ZAP: Killing Processes By Name

The `kill` command only terminates processes specified by process-id. This means you need to run `ps` to find the pid. Using `zap` can be dangerous, so better to run it interactively and use `pick` to select the victims.

IFS – Internal Field Separator

The shell variable IFS (internal field separator) is a string of characters that separate words in argument lists such as backquotes and for statements.

Usually, `$IFS` contains a blank, a tab, and a newline, but we can change it to anything, e.g. just a new line.

READ – Reading User Inpu

The shell built-in `read` command reads one line of text from the standard input and assigns the text (without the newline) as the value of the named variable.

```
read meep
> hello meep
echo $meep       //"hello meep"
```

The most common use of `read` is in .bash_profile to set up the environment when logging in, primarily to set shell vars like `TERM`.

`read` can only be read from the standard input. It can't even be redirected. None of the shell built-in commands (unlike things like `for`) can be redirected with `<` or `>`. So this doesn't work:

```
echo meep < /etc/passwd
```

If you use `echo -n`, it suppresses the final newline so the response can be typed on the same line as the prompt. And of course, prompts are printed on

`/dev/tty`.

Programming with Standard I/O

C is the standard language of UNIX systems – the kernal and all user programs are written in C.

Standard Input and Output: VIS

Many programs read only 1 input and write only 1 output.

`vis` copies its standard input to its standard output, except that it makes all non-printing characters visible by printing them as \nnn. where nnn is the octal value of the character.

Great for detecting strange characters that may have crept into files.

To scan multiple files:

```
cat file1 file2 | vis
```

`vis` was meant for non-text files.

The simplest input and output routines are called `getchar` and `putchar`. Each call to `getchar` gets the next character from the standard input, which may be a file or a pipe or the terminal (default) – the program doesn't know which.

Similarly, `putchar(a)` puts the character `a` on the standard output, which is also the terminal by default.

The function `printf` does output format conversion. Calls to `printf` and `putchar` can be intermixed in any order.

There is a corresponding function `scanf` for input format conversion - it will read the standard input and break it up into strings, numbers, etc, as desired. Calls to `scanf` and `getchar` can also be intermixed in any order.

PROBLEMS

- How to use the `&` sign
- How to use `here documents`.

Exercises

1. Modify `cal`
2. Use `for` to loop over `$PATH`, directories and filenames
3. Use all the operators in a program or over several programs.