Advanced Scientific Programming

Part 1 Linux and Shell

D. S. Hwang

Dankook University

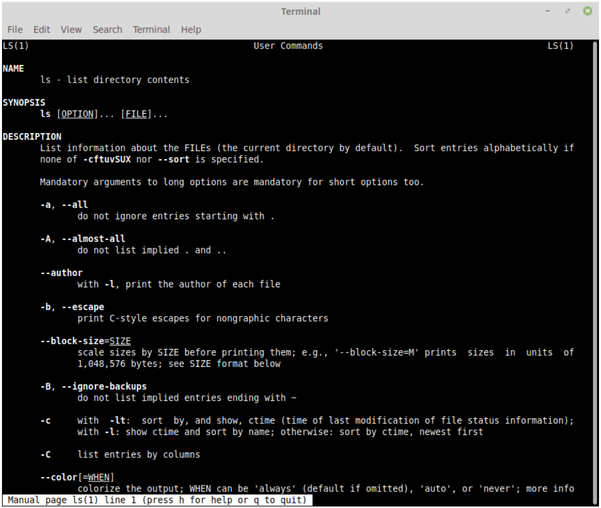
April 23, 2020

1. Introduction
2. Standard File System
3. Basic Command
4. File Permission and Globbing
5. Process
6. Shell Programming

The Linux Shell I

- Provides a command line interface(CLI) to the operating system
- Large variety of shells: `bash, tcsh, csh, ksh, zsh`
- Let us review commands for `bash`, the default on most systems
- Documentation can be found by typing `man <command>`, e.g. `man bash`

The Linux Shell II



```
LS(1)                                        User Commands                                       LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List information about the FILEs (the current directory by default).  Sort entries alphabetically if
       none of -cftuvSUX nor --sort is specified.

       Mandatory arguments to long options are mandatory for short options too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..

       --author
              with -l, print the author of each file

       -b, --escape
              print C-style escapes for nongraphic characters

       --block-size=SIZE
              scale sizes by SIZE before printing them; e.g., '--block-size=M' prints  sizes  in  units  of
              1,048,576 bytes; see SIZE format below

       -B, --ignore-backups
              do not list implied entries ending with ~

       -c     with  -lt:  sort  by, and show, ctime (time of last modification of file status information);
              with -l: show ctime and sort by name; otherwise: sort by ctime, newest first

       -C     list entries by columns

       --color[=WHEN]
              colorize the output; WHEN can be 'always' (default if omitted), 'auto', or 'never'; more info
 Manual page ls(1) line 1 (press h for help or q to quit)
```

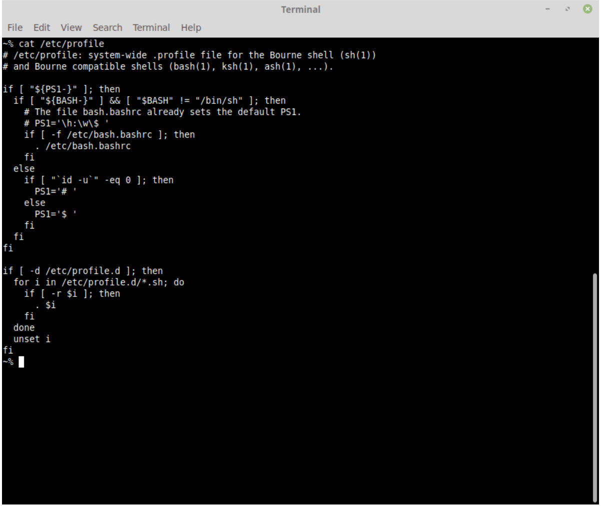
The Linux Shell III

- On login the system executes:
  - System file: /etc/profile, /etc/bashrc
  - For login shells, run ~/.bashrc
  - For non-login shells, run ~/.bashrc_profile
- Bash start-up files
  - /etc/profile sets an environment and defines start programs for all users
  - /etc/bashrc defines aliases and functions for all users
  - ~/.bash_profile sets an environment and defines start programs for a user
  - ~/.bashrc defines aliases and functions for a user
- To avoid complications it's recommended to have your ~/.bashrc_profile as a symlink to ~/.bashrc.
- The execution order for start-up files is /etc/profile, ~/.bash_profile, and ~/.bashrc.

The Linux Shell IV

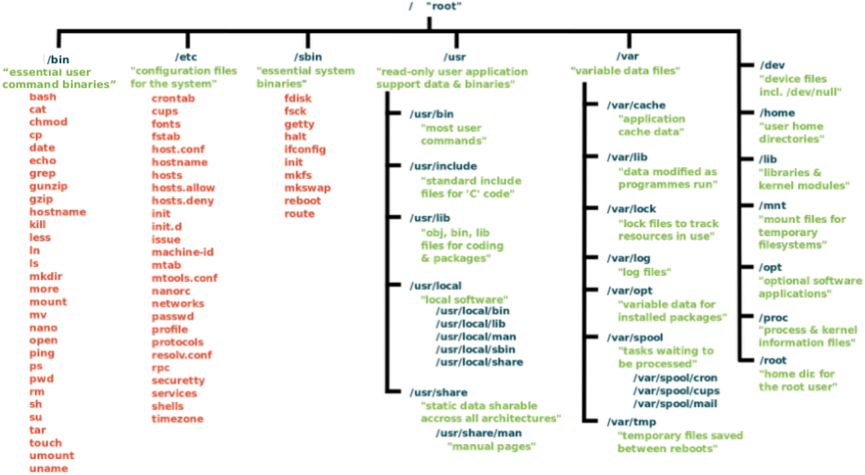
- The shell checks the first line of every program and if it finds `#!<interpreter>` it uses the the given interpreter to evaluate the file(e.g. `#!/usr/bin/env python`)
- `bash` is a very powerful shell that can be used for input redirection (`<,>, &, ...`), job control (fg, bg, jobs), file globbing (*, [0-9], ...).
- Check the `man` page for more!

The Linux Shell V



```
~% cat /etc/profile
# /etc/profile: system-wide .profile file for the Bourne shell (sh(1))
# and Bourne compatible shells (bash(1), ksh(1), ash(1), ...).

if [ "${PS1-}" ]; then
  if [ "${BASH-}" ] && [ "$BASH" != "/bin/sh" ]; then
    # The file bash.bashrc already sets the default PS1.
    # PS1='\h:\w\$ '
    if [ -f /etc/bash.bashrc ]; then
      . /etc/bash.bashrc
    fi
  else
    if [ "`id -u`" -eq 0 ]; then
      PS1='# '
    else
      PS1='$ '
    fi
  fi
fi

if [ -d /etc/profile.d ]; then
  for i in /etc/profile.d/*.sh; do
    if [ -r $i ]; then
      . $i
    fi
  done
  unset i
fi
~%
```

Standard File System I



/ "root"

/bin
"essential user command binaries"
bash
cat
chmod
cp
date
echo
grep
gunzip
gzip
hostname
kill
less
ln
ls
mkdir
more
mount
mv
nano
open
ping
ps
pwd
rm
sh
su
tar
touch
umount
uname

/etc
"configuration files for the system"
crontab
cups
fonts
fstab
host.conf
hostname
hosts
hosts.allow
hosts.deny
init
init.d
issue
machine-id
mtab
mtools.conf
nanorc
networks
passwd
profile
protocols
resolv.conf
rpc
securetty
services
shells
timezone

/sbin
"essential system binaries"
fdisk
fsck
getty
halt
ifconfig
init
mkfs
mkswap
reboot
route

/usr
"read-only user application support data & binaries"

/usr/bin
"most user commands"

/usr/include
"standard include files for 'C' code"

/usr/lib
"obj, bin, lib files for coding & packages"

/usr/local
"local software"
/usr/local/bin
/usr/local/lib
/usr/local/man
/usr/local/sbin
/usr/local/share

/usr/share
"static data sharable accross all architectures"
/usr/share/man
"manual pages"

/var
"variable data files"

/var/cache
"application cache data"

/var/lib
"data modified as programmes run"

/var/lock
"lock files to track resources in use"

/var/log
"log files"

/var/opt
"variable data for installed packages"

/var/spool
"tasks waiting to be processed"
/var/spool/cron
/var/spool/cups
/var/spool/mail

/var/tmp
"temporary files saved between reboots"

/dev
"device files incl. /dev/null"

/home
"user home directories"

/lib
"libraries & kernel modules"

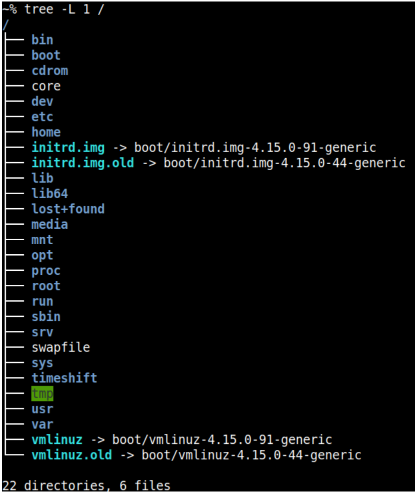
/mnt
"mount files for temporary filesystems"

/opt
"optional software applications"

/proc
"process & kernel information files"

/root
"home dir. for the root user"

Standard File System II

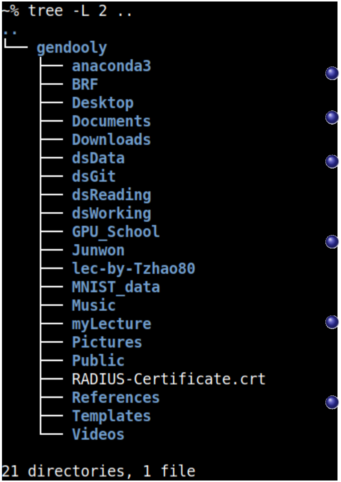


```
~% tree -L 1 /
/
├── bin
├── boot
├── cdrom
├── core
├── dev
├── etc
├── home
├── initrd.img -> boot/initrd.img-4.15.0-91-generic
├── initrd.img.old -> boot/initrd.img-4.15.0-44-generic
├── lib
├── lib64
├── lost+found
├── media
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin
├── srv
├── swapfile
├── sys
├── timeshift
├── tmp
├── usr
├── var
├── vmlinuz -> boot/vmlinuz-4.15.0-91-generic
└── vmlinuz.old -> boot/vmlinuz-4.15.0-44-generic

22 directories, 6 files
```

Standard File System III

- /bin and /usr/bin contain standard Linux commands
- /sbin and /usr/sbin contain system administration commands(`/sbin` stands for "safe" bin)
- /lib and /usr/lib contain standard Linux libraries
- /dev contains device drivers
- /var contains configuration and log files
- /etc contains default configuration files
- /usr/local/bin contains commands not a part of the distribution, added by the administrator on the installed Linux
- /opt contains commercial software
- /tmp stores temporary files

Home Directory I



```
~% tree -L 2 ..
..
└── gendooly
    ├── anaconda3
    ├── BRF
    ├── Desktop
    ├── Documents
    ├── Downloads
    ├── dsData
    ├── dsGit
    ├── dsReading
    ├── dsWorking
    ├── GPU_School
    ├── Junwon
    ├── lec-by-Tzhao80
    ├── MNIST_data
    ├── Music
    ├── myLecture
    ├── Pictures
    ├── Public
    ├── RADIUS-Certificate.crt
    ├── References
    ├── Templates
    └── Videos

21 directories, 1 file
```

- gendooly is my home directory.
- Desktop is the directory for the desktop.
- Documents is the default directory for documents.
- Downloads is the recommended directory for downloaded files.
- Public is the folder that is shared on File Sharing.
- Videos is the default directory for music and media files.

Basic Command I

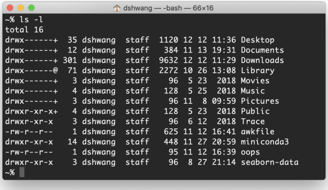
- `ls` lists directory contents, e.g. `ls -ltr`.
- `pwd` displays the current working directory, e.g. `pwd`
- `cd` changes a working directory, e.g. `cd ..`
- `cp` copies files, e.g. `cp -a /data/*.h5 /mnt/backup/data`
- `mv` moves files, e.g. `mv * /data`
- `rm` deletes files, e.g. `rm -rf /.*`
- `mkdir` creates directories, e.g. `mkdir -p /data/001-data`
- `cp` copies files, e.g. `cp -a /data/*.h5 /mnt/backup/data`
- `ln` creates symbolic links, e.g. `ln -s new /old`
- `grep` finds matched strings, e.g. `grep Result text.txt`
- `more` displays file contents, e.g. `more text.txt`
- `cat` displays file contents, e.g. `cat text.txt`

Basic Command II

- `tail` displays file contents, e.g. `tail -n 2 text.txt`
- `head` displays file contents, e.g. `head -n 2 text.txt`
- `wc` displays the number of lines, words, and bytes contained in files, e.g. `wc text.txt`
- `file` displays file types, e.g. `file text.txt`
- `env` displays environment variable, e.g. `env`
- `date` prints the current date and time on the screen, e.g. `date`
- `history` shows all the commands used in the past for the current terminal session., e.g. `history`
- `man` gets help on any command , e.g. `man ls`
- `clear` gives a clean window to work on , e.g. `clear`
- `users` displays the current users, e.g. `users`
- `groups` shows group memberships, e.g. `groups`

File Permission I

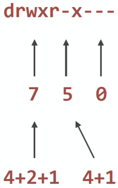
- In Linux systems, most things are represented by files.
- All files have owner, group and other permissions.
- The basic 3 permissions are read (4), write(2) and execute(1).
- Permissions can be changed with `chmod`, e.g. `chmod 760 text.txt`.
- Owners can be changed with `chown`, e.g. `chown Doly text.txt`
- Permissions of newly created files are determined by the users' `umask`



```
~% ls -l
total 16
drwx------+  35 dshwang  staff  1120 12 12 11:36 Desktop
drwx------+  12 dshwang  staff   384 11 13 19:31 Documents
drwx------+ 301 dshwang  staff  9632 12 12 11:29 Downloads
drwx------@  71 dshwang  staff  2272 10 26 13:08 Library
drwx------+   3 dshwang  staff    96  5 23  2018 Movies
drwx------+   4 dshwang  staff   128  5 25  2018 Music
drwx------+   3 dshwang  staff    96 11  8 09:59 Pictures
drwxr-xr-x+   4 dshwang  staff   128  5 23  2018 Public
drwxr-xr-x    3 dshwang  staff    96  6 12  2018 Trace
-rw-r--r--    1 dshwang  staff   625 11 12 16:41 awkfile
drwxr-xr-x   14 dshwang  staff   448 11 27 20:59 miniconda3
-rw-r--r--    1 dshwang  staff    95 11 12 16:39 oops
drwxr-xr-x    3 dshwang  staff    96  8 27 21:14 seaborn-data
~% 
```

File Permission II

- In Linux systems, most things are represented by files.
- All files and directories have owner(user), group and other permissions.
- The basic 3 permissions are read (4), write(2) and execute(1).
- Permissions can be changed with `chmod`, e.g. `chmod 760 text.txt`.
- Owners can be changed with `chown`, e.g. `chown Doly text.txt`
- Groups can be changed with `chgrp`, e.g. `chgrp GROUP2 text.txt`
- Permissions of newly created files are determined by the users' `umask`



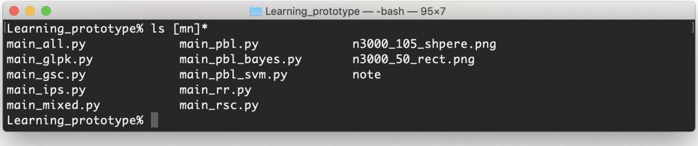
drwxr-x---

7 5 0

4+2+1 4+1

File Globbing I

- Linux shell can represent multiple filenames by using wild characters.
  - * matches 0 or more characters
  - ? matches exactly 1 character
  - [abc] matches 1 character in the set
  - [a-z] matches 1 character in the range



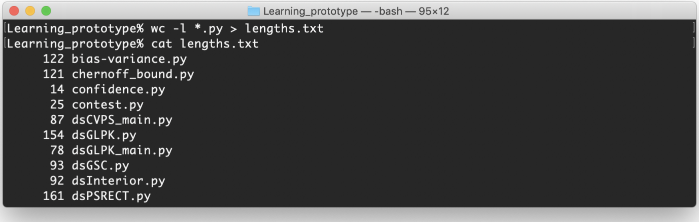
```
Learning_prototype% ls [mn]*
main_all.py             main_pbl.py             n3000_105_shpere.png
main_glpk.py            main_pbl_bayes.py       n3000_50_rect.png
main_gsc.py             main_pbl_svm.py         note
main_ips.py             main_rr.py
main_mixed.py           main_rsc.py
Learning_prototype%
```

Check it out: `man glob`

Pipes and Filters I

Linux assumes that all output is going to some kind of file. The screen is a file called `/dev/tty`.

- Use > to redirect the output of a command to a file or:



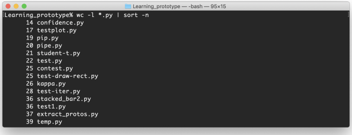
Learning_prototype — -bash — 95×12

```
Learning_prototype% wc -l *.py > lengths.txt
Learning_prototype% cat lengths.txt
     122 bias-variance.py
     121 chernoff_bound.py
      14 confidence.py
      25 contest.py
      87 dsCVPS_main.py
     154 dsGLPK.py
      78 dsGLPK_main.py
      93 dsGSC.py
      92 dsInterior.py
     161 dsPSRECT.py
```

- Use >> works similarly, but the output is appended to the file instead of replacing it.

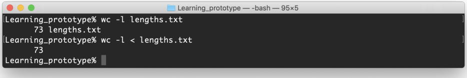
Pipes and Filters II

- Use | (called a pipe) to mash-up two or more commands at the same time and run them consecutively



```
Learning_prototype% wc -l *.py | sort -n
      14 confidence.py
      17 testplot.py
      19 pip.py
      20 pipe.py
      21 student-t.py
      22 test.py
      25 contest.py
      25 test-draw-rect.py
      26 kappa.py
      28 test-iter.py
      36 stacked_bar2.py
      36 test1.py
      37 extract_protos.py
      39 temp.py
```

- Use < to make the content of a file the input of a command



```
Learning_prototype% wc -l lengths.txt
      73 lengths.txt
Learning_prototype% wc -l < lengths.txt
      73
Learning_prototype% 
```

Most commands that read from standard input also accept files as arguments so this is not as useful.

Standard File Descriptors I

Make sure that everything in Linux is a file. Moreover, every file has its own file descriptor.

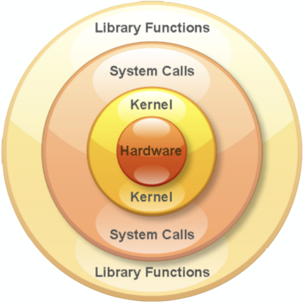
- Every process opens 3 numbered standard file descriptors, `stdin (0)`, `stdout (1)`, `stderr (2)`.
- They correspond to the input (stdin), output (stdout) and error messages (stderr).

```
% pgm 2> error.log

% find . −name mine∗ 2> error.log

# list directories and store both error
#  and standard output into a file
% ls Documents > dir.lst 2>&1
```

Overview

Process

- A process is a running program and has a unique pid (process id).
- Each process has a parent. It inherits its environment from the parent.
- Use pstree to show running processes as a tree.
- Each process has its own memory address.
- Processes can have one or more threads.
- All threads in a process share the same memory space.



Process Environment I

Process

Environment variables are dynamic values which affect the processes or programs on a computer. That way, environment variables change the way a software/programs behave.

- Common environment variables:

Variable	Description	Usage
PATH	A colon (:)-separated list of directories of executable files.	`echo $PATH`
USER	The username	`echo $USER`
HOME	Default path to the user's home directory	`echo $HOME`
EDITOR	Path to the editor	`echo $EDITOR`
UID	User's unique ID	`echo $UID`
TERM	Default terminal emulator	`echo $TERM`
SHELL	Shell being used by the user	`echo $SHELL`
MAIL	Default mail box path	`echo $MAIL`
HOSTNAME	Default mail box path	`echo $HOSTNAME`

Process Environment II

Process

- Each process has a parent process from which it inherits the environment.
- PATH is an environment variables used to search for executables. Display it with `echo $PATH`.
- Change variables using `export`, e.g. `export PATH=~/bin:$PATH`
- Use `which` to find out the full path of an executable, e.g. `which cp`
- Create a user defined variable, e.g. `VAR_NAME = value`
- Delete environmental variable, e.g. `unset VAR_NAME`
- `LD_LIBRARY_PATH` is another, which lets the program know where to find dynamic libraries.
- Use `ldd` to print out the libraries found by the system:

Process Environment III

Process

```
~% otool -L /bin/cat # ldd for linux
/bin/cat:
        /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1252.200.5)
```

Try `man environ` for more information

Job Control

Process

- Signals can be used to control a process.
- `Ctrl+C`(SIGINT) asks the process to terminate.
- `Ctrl+Z`(SIGSTP) suspends the process. Use fg to continue the process.
- Use the kill command to send signals.
- `kill 9 <pid>` sends the SIGKILL signal to the process causing it to terminate immediately.
- Certain processes (stuck inside kernel calls) cannot be killed.
- Use nice to control the priority of execution of the process.

Check `man signal` for more information.

Overview

Shell Programming

- Understand some of the basics of shell script programming (aka shell scripting)
- Introduce some of the possibilities of simple but powerful programming available under the Bourne shell(`bash`)
- Understanding of some Linux commands, and getting used to some of the more common ones
- A shell script combines Linux commands to perform a specific task.
- All shell scripts operate similar to those found in other programming languages.
- `man bash` or `https://www.gnu.org` for more information

Shell basic

Shell Programming

A simple shell script displays "Hey, Dankook!"

```
% echo '#!/bin/sh' > my-script.sh
% echo 'echo Hey, Dankook!' >> my-script.sh
% chmod 755 my-script.sh
% ./my-script.sh
Hey, Dankook!
```

- The first line tells Linux that the file is to be executed by `/bin/sh`.
- In Linux, `/bin/sh` is normally a symbolic link to `bash`.
- # marks the line as comment.
- echo, with two parameters, or arguments - the first is "Hey,"; the second is "Dankook!".

Display all of the current shell variables in alphabetical order

```
% set | less
ADDR2LINE=/home/gendooly/anaconda3/bin/x86_64-conda_cos6-linux-gnu-addr2line
AR=/home/gendooly/anaconda3/bin/x86_64-conda_cos6-linux-gnu-ar
AS=/home/gendooly/anaconda3/bin/x86_64-conda_cos6-linux-gnu-as
```

Echo

Shell Programming

```
echo [-ne] str1 str2 ...
```

- A simple shell command consists of the command itself followed by arguments, separated by spaces.
- echo outputs the list of arguments to the standard output.
- `-n` ignores new line.
- `-e` runs escape characters.

Quoting

Shell Programming

- Quoting is used to remove the special meaning of certain characters or words to the shell.
  - disable special treatment for special characters
  - prevent reserved words from being recognized
  - disable parameter expansion
- Each of the shell metacharacters has special meaning to the shell and must be quoted if it is to represent itself.
- Singe quote is used to prevent the shell from interpreting any special characters.
- Double quote allows the shell to interpret : $, ’ , and ”
- Back quote forces the execution of the commands and the result is assigned to the variable.

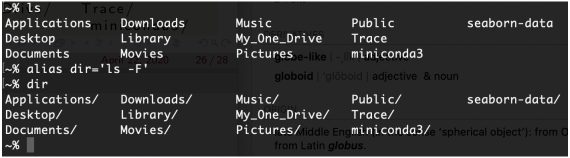
```
temp% today=‘date ”+%Y-%m-%d %H:%M:%S” ‘
temp% echo $today
2020-04-23 10:37:55
```

Alias

Shell Programming

```
alias var=string
```

- Run any command or a ordered list of commends with a user-defined string
- Very useful to define customized commands



```
~% ls
Applications    Downloads       Music           Public          seaborn-data
Desktop         Library         My_One_Drive    Trace
Documents       Movies          Pictures        miniconda3
~% alias dir='ls -F'
~% dir
Applications/   Downloads/      Music/          Public/         seaborn-data/
Desktop/        Library/        My_One_Drive/   Trace/
Documents/      Movies/         Pictures/       miniconda3/
~%
```

History

Shell Programming

`alias h=history`

- List the input commands logged into `~/.bash_history`
- Very useful to define customized commands
- `HISTSIZE` resets the number of logged commands

```
~% HISTSIZE=5
~% alias h=history
~% h
497  alias h=history
498  h
499  HISTSIZE=5
500  alias h=history
501  h
```

Variables I

Shell Programming

```
var=string
var=cmd_string
```

- Variables are set through defining shell variables.
- Variables are named with any set of alphabetic characgters including '_'.
- There must be no spaces around the $=$ sign.
- $ is prefixed to retrieve variable values: `$var`.

```
#!/bin/sh
MY_MESSAGE="Hey, Dankook!"
echo $MY_MESSAGE
```

Variables II

Shell Programming

Set variable names using the `read` command:

```
#!/bin/sh
echo What is your name?
read MY_NAME
echo "Hello $MY_NAME - hope you're well."
```

Period(.) intructs Linux to execute a script.

```
temp% cat test.sh
#! /bin/bash
MY_MESSAGE="Hey, Dankook!"
echo $MY_MESSAGE
temp% . test.sh
Hey, Dankook!
```

Variables III

Shell Programming

Shell script with arguments:

- $$ Shell process id
- $0 Shell script name
- $ 0 ∽ 9,${10} Script argument for a shell command
- $* The list of arguments
- $# The number of arguments
- $? The exit coder of the last command: 0(Ture) or 1(False)

```
temp% cat test.sh
#! /bin/bash
echo $1 $2
echo "Procees id:" $$
echo "file name: " $0
echo $# " arguments"
echo "arguments : " $*
```

Variables IV

Shell Programming

```
temp% . test.sh Dankook University!
Dankook University!
Procees id: 1134
file name: -bash
2 arguments
arguments : Dankook University!
```

Make a script executable:

```
temp% chmod u+x test.sh

temp% ./test.sh Dankook University!
Dankook University!
Procees id: 1321
file name: ./test.sh
2 arguments
arguments : Dankook University!
```

List Variables I

Shell Programming

```
name=(str1 str2 str3 ... )
```

- A variable stores a set of values.
- Each value is referred by an index starting 0.
  - `$name[i]` for the i-th value
  - `$name[*]` and `$name[@]` for all the values
  - `$#name[*]` and `$#name[@]` for the number of values

List Variables II

Shell Programming

```
temp% clist=(cat dog bear rabbit)
temp% echo ${clist[1]}
dog
temp% echo ${clist[3]}
rabbit
temp% echo ${clist[3]} ${clist[2]}
rabbit bear
temp% echo ${clist[*]}
cat dog bear rabbit
temp% echo ${clist[@]}
cat dog bear rabbit
temp% echo ${#clist[@]}
4
temp% echo ${#clist[*]}
4
```

Standard Input

Shell Programming

`read name1 name2 ....`

- Allow a user to input values for variables
- Each input variable is assigned to a word of an input string from left to right.
- The last variable holds the rest of the input string.

```
temp% read X Y
Dankook University!
temp% echo $X
Dankook
temp% echo $Y
University!
temp% echo $X $Y
Dankook University!
```

Command test

Shell Programming

```
test expr
( test expr )
[ expr ]
((  expr ))
```

- Provides no output, but returns an exit status of 0 for ”true” (test successful) and 1 for ”false”(test failed)
- Frequently used as part of a conditional expression

```
temp% [ $USER = dshwang ]
temp% echo $?
0
temp% test $USER = dshwang
temp% echo $?
0
temp% test $USER = dshwan
temp% echo $?
1
```

Command if-then-else-fi I

Shell Programming

```
if [ expr ]
    then
         stmt1
else
         stmt2
fi
```

- Provides no output, but returns an exit status of 0 for "true" (test successful) and 1 for "false"(test failed)
- Frequently used as part of a conditional expression

Command `if-then-else-fi` II

Shell Programming

```
temp% cat dswc.sh
#!/bin/bash

if [ $# -eq 1 ]; then
    wc $1
else
    echo Usage: $0 file
fi
temp% . dswc.sh
Usage: -bash file
temp% chmod +x dswc.sh
temp% ./dswc.sh dswc.sh
7       16      78 dswc.sh
temp% ./dswc.sh
Usage: ./dswc.sh file

```

Command `if-then-else-fi` III

Shell Programming

```
temp% num=4; if (test $num -gt 5); then echo "yes"; else echo "no"; fi
no
temp% num=4; if test $num -gt 5; then echo "yes"; else echo "no"; fi
no
temp% num=4; if [ $num -gt 5 ]; then echo "yes"; else echo "no"; fi
no
temp% num=4; if (($num > 5)); then echo "yes"; else echo "no"; fi
no

```

Command if-then-else-fi IV

Shell Programming

```
temp% cat dscount.sh
#!/bin/bash
# count the number of subdirectories in a given directory

if [ $# -eq 0 ];
then
        dir="."
else
        dir=$1
fi

echo -n 'the number of subdirectories in $dir:'
ls $dir | wc -l
temp% ./dscount.sh
the number of subdirectories in $dir:        4
```

Nested `if-then-elif-then-else-fi` I

Shell Programming

```
if [ expr1 ]; then
     stmt1
elif [ expr2 ]; then
    stmt2
else
    stmt3
fi
```

- Use nested conditional statements

Nested if-then-elif-then-else-fi II

Shell Programming

```
temp% . dsscore.sh
subject score:93
Your credit is A.
temp% cat dsscore.sh
#!/bin/bash
# map scores to credits

echo —n ”subject score:”
read score
if (( $score >= 90 )); then
        credit=A
elif (( $score >= 80 )); then
        credit=B
else
        credit=F
fi
echo —n ”Your credit is $credit.”
echo
```

Statement case-esac I

Shell Programming

```
case var in
   val1) stmt1;;
   val2) stmt2;;
     .....
   *) stmtk;;
esac
```

- Use multiple choice statement instead of nested conditional statements

Statement case-esac II

Shell Programming

```
temp% cat dsscore2.sh
#!/bin/bash
# map scores to credits

echo -n "subject score:"
read score
let grade=$score/10
case $scire in
    "10"|"9") echo A;;
    "8") echo B;;
    "7") echo C;;
    *) echo F;;
esac
temp% . dsscore2.sh
subject score:80
F
```

Statement for-done I

Shell Programming

```
for var in list; do
   stmt1
   stmt2
     ...
done
```

- Repeat the group of statements for each list value

Statement while-done I

Shell Programming

```
while expr; do
    stmt1
    stmt2
     ...
done
```

- Repeat the while body while expr is true

Statement while-done II

Shell Programming

```
temp% cat dssum.sh
#!/bin/bash

let i=0
let k=10
let sum=0

while(( $i <= $k )); do
    let sum=sum+$i
    let i+=1
done
echo ”sum = $sum”
temp% . dssum.sh
sum = 55
```

Using Function I

Shell Programming

```
ftn_name() {
    stmt1
    stmt2
     ....
}
```

- Define functions for reuse
- Bash functions do not retun a value to a caller.
- Set a global variable to return values or use command substitution
- Funciton arguments are the same as ones on the command line instruction.

Using Function II

Shell Programming

```
temp% cat dsfunction.sh
#!/bin/bash

lshead(){
    echo "argument $1"
    echo "choose the three files in $1"
    ls -1 $1 | head -3
}

lshead .
temp% . dsfunction.sh ..
argument .
choose the three files in .
dscheck.sh
dscount.sh
dsfor.sh
```

Using Function III

Shell Programming

- The callee assigns the output to the global variable.

```
temp% cat dsfun1.sh
#!/bin/bash
# use a global variable

function func(){
    result='12345'
}

func
echo "$result"
temp% sh dsfun1.sh
12345
```

Using Function IV

Shell Programming

- Return the result to the caller by substitution
- The result becomes the output to stdio and the caller uses command substitution.

```
temp% cat dsfun2.sh
#!/bin/bash
# use a local variable

function func(){
    local result='12345'
    echo "$result"
}

result=$(func)
echo "$result"
temp% sh dsfun2.sh
12345
```

Expression I

Shell Programming

Arithematic comparison operator:

- var1 -eq var2 returns 0 if var1 $=$ var2 otherwise 1
- var1 -ne var2 returns 0 if var1 $! =$ var2 otherwise 1
- var1 -gt var2 returns 0 if var1 $>$ var2 otherwise 1
- var1 -ge var2 returns 0 if var1 $\geq$ var2 otherwise 1
- var1 -lt var2 returns 0 if var1 $<$ var2 otherwise 1
- var1 -le var2 returns 0 if var1 $\leq$ var2 otherwise 1

Expression II

Shell Programming

```
temp% cat dscount.sh
#!/bin/bash
# count the number of subdirectories in a given directory

if [ $# -eq 0 ]; then
        dir="."
else
        dir=$1
fi

echo -n 'the number of subdirectories in $dir:'
ls $dir | wc -l
temp% ./dscount.sh
the number of subdirectories in $dir:       4
```

Expression III

Shell Programming

String comparison operator:

- str1 $==$ str2 returns 0 if str1 $=$ str2 otherwise 1
- var1 $! =$ var2 returns 0 if var1 $! =$ var2 otherwise 1
- -n var returns 0 if var $! =$ NULL otherwise 1
- -z var returns 0 if var $=$ NULL otherwise 1
- var1 -lt var2 returns 0 if var $! =$ NULL otherwise 1
- var1 -le var2 returns 0 if var1 $\leq$ var2 otherwise 1

Expression IV

Shell Programming

```
temp% cat dsinput.sh
#!/bin/bash
# check continue or quit

echo -n 'continue[y/n]'
read reply
if [ $reply == 'y' ]; then
	echo '.... continue ....'
else
	echo '.... quit ....'
fi
temp% . dsinput.sh
continue[y/n]n
.... quit ....
```

Expression V

Shell Programming

File related operator:

- -a fname or -e fname returns 0 if fname exists otherwise 1
- -r fname returns 0 if fname is readable otherwise 1
- -w fname returns 0 if fname is writable otherwise 1
- -x fname returns 0 if fname is executable otherwise 1
- -o fname returns 0 if a user owns fname otherwise 1
- -z fname returns 0 if fname size is 0 otherwise 1
- -f fname returns 0 if fname is a general file otherwise 1
- -d fname returns 0 if fname is a directory otherwise 1

Expression VI

Shell Programming

```
temp% cat dscheck.sh
#!/bin/bash
# check the existence of a file

echo 'Input your file...'
read file

if [ -e $file ]; then
    wc $file
else
    echo "$file does not exits!!"
fi
temp% . dscheck.sh
Input your file...
dscheck.sh
12      29      158 dscheck.sh
temp% . dscheck.sh
Input your file...
xyz
xyz does not exits!!
```

Expression VII

Shell Programming

Bool operator:

- ! Negation
- && Logical and
- || Logical or

```
temp% if [ -f dscount.sh ] && [ -w dscount.sh ]; then uptime; fi
20:10  up  6:13, 2 users, load averages: 1.05 1.15 1.17
```

Command let

Shell Programming

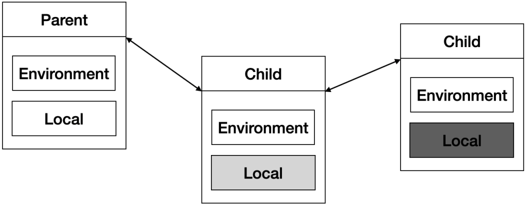
```
let var=expr
```

- Bash script usually deals with strings.
- There is a way to compute simple arithmatic expressions.
- `var` stores the result of `expr`.

```
temp% let a=12*11
temp% echo $a
132
temp% let a++
temp% echo $a
1321
temp% let $a
temp% echo $a
1321
temp% let a=a+1
temp% echo $a
1322
```

Local vs Environment Variable

Shell Programming



- The parent shell can initialize a child shell if neccesary.
- The environment variables are inherited to a child shell.
- A child shell defines local variables for use.

References

1. https://www.shellscript.sh/variables1.html
2. Mokhatar Ebrahim and Andrew Mallett, Mastering Linux Shell Scripting, Packt, 2018
3. Brian Ward, How LINUX works: What every superuser should know, No Starch Press, 2004