Curtin College

in association with

Curtin University

Shell environment & Scripts

Computer Systems 2000 (CS2000)

Trimester 2 2020

Why?

- A Bash script is a plain text file which contains a series of commands.
  - These commands are a mixture of commands we would normally type on the command line (such as ls or cp for example) and commands we could type on the command line but generally wouldn't
  - Anything you can run normally on the command line can be put into a script and it will do exactly the same thing. Similarly, anything you can put into a script can also be run normally on the command line and it will do exactly the same thing.
  - Note: Output from one command can be used as the input to the next
- Automation
  - Repetitive tasks
  - Complex command structure
  - Save typing
  - Don't have to remember!

When NOT to use Shell Scripts

- Resource-intensive tasks, especially where speed is a factor(sorting, hashing, recursion)
- Procedures involving heavy-duty maths operations, especially floating point arithmetic, arbitrary precision calculations, where structured programming isa necessity (type-checking of variables, function prototypes, etc.)
- Situations where security is important
- Project consists of subcomponents with interlocking dependencies

When NOT to use Shell Scripts

- Extensive file operations required (Bash is limited to serial file access, and that only in a particularly clumsy and inefficient line-by-line fashion.)
- Need data structures, such as linked lists or trees
- Need to generate / manipulate graphics or GUIs
- Need direct access to system hardware or port / socket I/O
- Need to use libraries or interface with legacy code
- Proprietary, closed-source applications

Shell Environment

- Shell environment
  - Consists of a set of variables with values.
  - These values are important information for the shell and the programs that run from the shell.
  - You can define new variables and change the values of the variables.
- Example: PATH determines where the shell looks for the file corresponding to your command.
- Example: SHELL indicates what kind of shell you are using.

Shell Variables

- How do we use the values in the shell variables ?????
  - Put a $ in front of their names.
    - e.g: echo $HOME
    - Prints the value that is stored in the variable HOME.
- Many are defined in .cshrc and .login for the C shell and in .bashrc and .bash_profile for bash.

Shell Variables

- Example .bashrc file:

```
#Global variables here
export PATH TERM HOME HISTFILE
export PATH=$PATH:/usr/student/bin:/usr/
local/bin:/usr/
local/sbin:/usr/local/X11/bin:/usr/sbin:/
usr/bin:.:~/bin:
/usr/local/j2se/bin
#some nice aliases here
export PS1='[\u@\h \W]$ '
export PRINTER=ps2
```

Shell Variables

- Two kinds of shell variables:
  - Environment variables
    - Available in the current shell and the programs invoked from the shell
  - Regular shell variables
    - Not available in programs invoked from this shell.
- Comments on examples:
  - Examples are shown for both the C shell as well as for bash.
    - echo $SHELL

Shell Variables

- To explicitly invoke a particular shell, type the name of the shell on the command line:
  - E.g:
  - $ csh ! invokes the C shell.
  - $ bash ! invokes bash.
- Declaring regular variables in the C shell:
  - set varname = varvalue
  - Space between varname and varvalue is optional.
  - Sets the variable varname to have value varvalue.

Shell Variables

- Declaring regular variables in bash:
  - varname=varvalue
  - No space between varname and varvalue.
  - Sets the variable varname to have value varvalue.

```
[div-boslap:~] iainmurray% csh
% set test = "this is a test"
% echo $test
this is a test
% echo test
test

[div-boslap:~] iainmurray% bash
bash-3.2$ test="this is a test"
bash-3.2$ echo $test
this is a test
bash-3.2$ echo test
test
bash-3.2$
```

Shell Variables

- Remove declaration of regular variables:
  - Use the unset command
    - Works for both the C shell and bash
    - unset varname
    - Once variable is unset, the value that previously was assigned to that variable does not exist anymore.

Example

```
bash-3.2$ test="this is a test"
bash-3.2$ echo $test
this is a test
bash-3.2$ unset test
bash-3.2$ echo $test

bash-3.2$
```

NOTE: Once the variable var is unset, there is no value as part of the variable. Not true in all shells.

Shell Variables

- Declaring environment variables in the C shell:
  - setenv varname varvalue
  - Sets the environment variable varname to have value varvalue.
  - Notice that there is no ‘=’
  - Space between varname and varvalue is necessary.
- Declaring environment variables in bash:
  - Using the export command.
  - To change a regular variable to an environment variable, need to export them.
  - varname=varvalue
  - export varname
  - Sets the environment variable varname to have value varvalue.

Shell Variables

Example:

```
[div-boslap:~] iainmurray% csh
% setenv test "this is a test"
% echo $test
this is a test

[div-boslap:~] iainmurray% bash
bash-3.2$ test="this is a test”
bash-3.2$ export test
bash-3.2$ export test="this is a test"
bash-3.2$
```

NOTE: The declaration with the export command can be combined into one statement as shown.

Shell Variables

- Remove declaration of environment variables in the C shell:
  - Use the unsetenv command.
    - **unsetenv *varname***
  - Once variable is unset, the value that previously was assigned to that variable does not exist anymore.

Example

```
[div-boslap:~] iainmurray% csh
% setenv test "this is a test"
% echo $test
this is a test
% unsetenv test
% echo $test
test: Undefined variable.
%
```

NOTE: test is undefined as it has been unset.

Shell Variables

- Remove declaration of environment variables in bash:
  - Use the unset command.
  - unset varname
  - Once variable is unset, the value that previously was assigned to that variable does not exist anymore.

Shell Variables

```
[div-boslap:~] iainmurray% bash
bash-3.2$ var="this is a test"
bash-3.2$ export var
bash-3.2$ echo $var
this is a test
bash-3.2$ unset var
bash-3.2$ echo $var
bash-3.2$
```

Shell Variables

- We can use regular variables, just like environment variables, so why have environment variables ???
  - Regular variables are only available to the current shell.
  - Environment variables are accessible across shells and to all running programs.
    - What does this mean? example follows.

```
[bash-3.2$ var="testing the variables"
bash-3.2$ echo $var
testing the variables
bash-3.2$ bash
bash-3.2$ echo $var
bash-3.2$
```

NOTE: with the command bash, I invoke a new shell (bash) and in this shell, my variable var is not accessible anymore

```
bash-3.2$ echo $var
testing the variables
bash-3.2$ export var
bash-3.2$ bash
bash-3.2$ echo $var
testing the variables
bash-3.2$
```

NOTE: the environment variable is accessible even when I invoke another shell using the command bash.

Variables

```
[Iains-15-macbook-pro:~] iain% bash
bash-3.2$ var=123
bash-3.2$ echo $var
123
bash-3.2$ bash
bash-3.2$ echo $var

bash-3.2$ exit
exit
bash-3.2$ echo $var
123
bash-3.2$
```

Note: exiting one shell, variables from prior shell are still available

Shell Variables

- Common shell variables:
  - SHELL: the name of the login shell of the user.
  - PATH: the list of directories searched to find executables to execute.
  - MANPATH: where man looks for man pages.
  - LD_LIBRARY_PATH: where libraries for executables exist.
  - USER: the user name of the user who is logged into the system.
  - HOME: the user’s home directory.
  - MAIL: the user’s mail directory.
  - TERM: the kind of terminal the user is using.
  - DISPLAY: where X program windows are shown.
  - HOST: the name of the machine logged on to.
  - REMOTEHOST: the name of the host logged in from.

Shell Variables

- Quotes in Unix have a special meaning
  - Single quotes:
    - Stops shell variable expansion.
  - Back quotes:
    - Replace the quotes with the result of the execution of the command.

Example

Single Quotes

```
bash-3.2$ echo "Welcome $USER"
Welcome iainmurray
bash-3.2$ echo 'Welcome $USER'
Welcome $USER
bash-3.2$

iainmurray% set var = `hostname`
iainmurray% echo $var
div-boslap.eng.cage.curtin.edu.au
iainmurray%
```

NOTE: The hostname command returns the name of the machine, which in this case is div-boslap.eng.cage.curtin.edu.au

Shell Variables

- What about double quotes “ ” ?
  - No difference if they are used or not.

```
[div-boslap:~] iainmurray% echo Welcome
$USER
Welcome iainmurray
[div-boslap:~] iainmurray% echo "Welcome
$USER"
Welcome iainmurray
[div-boslap:~] iainmurray%
```

Shell Startup

- When csh and tcsh are executed, they read and run certain configuration files:
  - .login: run once when you log in
    - Contains one time initialisation, like TERM, HOME etc.
  - .cshrc: run each time another csh/tcsh process is invoked.
    - Sets variables, like PATH, HISTORY etc.
    - Aliases are normally written in this file.
- When bash is executed, it reads and runs certain configuration files:
  - .profile/.bash_profile: runs when you log in.
    - Contains one time initialisation, like TERM, HOME etc.
  - .bashrc: run each time another bash process is invoked.
    - Sets variables, like PATH, HISTORY etc.

Shell Startup

- Only modify the lines that you fully understand!
- Can cause errors if not careful.
- E.g:
    - alias ls=‘exit’

```
if ((-e /sitedep/LINUX))
        then
        source .login.linux
endif
setenv MAIL /usr/spool/
mail/$USER
setenv EXINIT 'set redraw
wm=8'
mesg y
set prompt = "* Hello *> “
logout
```

Shell Startup

- These files can be used for writing very useful commands.
  - Setting aliases.
  - Setting environment variables.
  - System setup.
  - Setting prompt.
  - Etc.

Scripting Outline

- Shell scripts.
    - Definition.
    - Uses of shell scripts.
    - Writing shell scripts.

Shell Scripts

- Shell scripts usually begin with a #! and a shell name (complete pathname of shell).
  - Pathname of shell be found using the which command.
  - The shell name is the shell that will execute this script.
    - E.g: #!/bin/bash
- If no shell is specified in the script file, the default is chosen to be the currently executing shell.

Shell Scripts

- Any Unix command can go in a shell script
  - Commands are executed in order or in the flow determined by control statements.
- Different shells have different control structures
  - The #! line is very important.
  - We will write shell scripts with the Bourne (again) shell (bash).

Shell Scripts

- A shell script is an executable program:
  - Must use chmod to change the permissions of the script to be executable

- Can run script explicitly also, by specifying the shell name.
  - E.g: $ bash myscript
  - E.g: $ csh myscript

- Consider the example
  - $ bash myscript
  - Invokes the bash shell and then runs the script using it.
  - myscript need not be an executable as bash is running the script on its behalf.

Shell Scripts

- Why write shell scripts ?
  - To avoid repetition:
    - If you do a sequence of steps with standard Unix commands over and over, why not do it all with just one command?
  - To automate difficult tasks:
    - Many commands have subtle and difficult options that you don’t want to figure out or remember every time .

Simple Example

- To run the script:
  - Step 1:
    - $ chmod u+x myscript
  - Step 2:
  - Run the script:
    - $ ./myscript
- Each line of the script is processed in order.

```
bash-3.2$ cat myscript
#! /bin/bash
rm -rf $HOME/.netscape/cache
rm -f $HOME/.netscape/his*
rm -f $HOME/.netscape/cookies
rm -f $HOME/.netscape/lock
rm -f $HOME/.netscape/.nfs*
rm -f $HOME/.pine-debug*
rm -fr $HOME/nsmail
```

Shell Variables

- Shell variables:
  - Declared by:
  - varname=varvalue
  - To make them an environment variable, we export it. export varname=varvalue
- Assigning the output of a command to a variable:
  - Using backquotes, we can assign the output of a command to a variable:

```
#! /bin/bash
filelist=`ls`
echo $filelist
```

Shell Scripts

- The expr command:
  - Calculates the value of an expression.

```
bash-3.2$ value=`expr 1 + 2`
bash-3.2$ echo $value
3
bash-3.2$
```

- Why is expr necessary?

```
bash-3.2$ file=1+2
bash-3.2$ echo $file
1+2
```

Notes on expr

- Variables as arguments:

```
bash-3.2$ count=5
bash-3.2$ count=`expr $count + 1`
bash-3.2$ echo $count
6
bash-3.2$
```

expr supports the following operators:

arithmetic operators
+   -   *   /   %
comparison operators:
<   <=   ==   !=   >=   >
boolean/logical operators:
& |
parentheses: ( )
precedence is the same as C, Java

Control Statements

- The three most common types of control statements:
  - conditionals
    - if/then/else, case, ...
  - loop statements
    - while, for, until, do, ...
  - branch statements
    - subroutine calls (good programming practice), goto (usage not recommended).

For Loops

- for loops allow the repetition of a command for a specific set of values.
- Syntax:

```
for var in value1 value2 ...
do
    command_set
done
```

  - command_set is executed with each value of var (value1, value2, ...) in sequence

NOTE: * is a wild card that stands for all files in the current directory

```
#! /bin/bash
for i in *
do
    echo $i
done
```

Conditionals

- Conditionals are used to “test” something.
  - In Java or C, they test whether a Boolean variable is true or false.
  - In a Bourne shell script, the only thing you can test is whether or not a command is “successful”.
- Every well behaved command returns back a return code.
  - 0 if it was successful
  - Non-zero if it was unsuccessful (actually 1..255)

The IF Command

- Simple form:

```
if decision_command_1
then
        command_set_1
fi
```

- Importance of having then on the next line:
  - Each line of a shell script is treated as one command.
  - then is a command in itself
  - Even though it is part of the if structure, it is treated separately.

Example

grep returns 0 if it finds something
returns non-zero otherwise

```
if grep unix myfile >/dev/null
then
        echo "It's there"
fi
```

redirect to /dev/null so that "intermediate" results do not get printed

Using *ELSE* with *IF* and *ELIF*

```
#! /bin/bash
if grep "UNIX" myfile >/dev/null
then
        echo UNIX occurs in myfile
else
        echo No!
        echo UNIX does not occur in myfile
Fi
```

```
#! /bin/bash
if grep "UNIX" myfile >/dev/null
then
        echo UNIX occurs in myfile
elif grep “DOS” myfile > /dev/null
then
        echo DOS appears in myfile not UNIX
else
        echo nobody is here in myfile
fi
```

Using Colon in Shell Scripts

- Sometimes, we do not want a statement to do anything.
  - In that case, use a colon ‘:’

```
if grep UNIX myfile > /dev/null
then
        :
fi
```

  - Does not do anything when UNIX is found in myfile.

The Test Command

- Use for checking validity.
- Three kinds:
  - Check on files.
  - Check on strings.
  - Check on integers
- Testing on files.
  - test –f file: does file exist and is not a directory?
  - test -d file: does file exist and is a directory?
  - test –x file: does file exist and is executable?
  - test –s file: does file exist and is longer than 0 bytes?

Example

```
#!/bin/bash
count=0
for i in *; do
if test –x $i
then
        count=`expr $count + 1`
fi
done
echo Total of $count files executable
```

NOTE: expr $count + 1 serves the purpose of count++

Notes on Test

- Testing on strings.
- test –z string: is string of length 0?
- test string1 = string2: does string1 equal string2?
- test string1 != string2: not equal?

```
#! /bin/bash
if test -z $REMOTEHOST
then
:
else
        DISPLAY="$REMOTEHOST:0“
        export DISPLAY
fi
```

NOTE: This example tests to see if the value of REMOTEHOST is a string of length > 0 or not, then sets the DISPLAY to the appropriate value.

Notes on Test

- Testing on integers.
  - test int1 –eq int2: is int1 equal to int2 ?
  - test int1 –ne int2: is int1 not equal to int2 ?
  - test int1 –lt int2: is int1 less than to int2 ?
  - test int1 –gt int2: is int1 greater than to int2 ?
  - test int1 –le int2: is int1 less than or equal to int2 ?
  - test int1 –ge int2: is int1 greater than or equal to int2 ?

Example

- The test command has an alias ‘[]’.
  - Each bracket must be surrounded by spaces

```
#!/bin/bash
smallest=10000
for i in 5 8 19 8 7 3
  do
      if test $i -lt
      $smallest
      then
            smallest=$i
      fi
done
echo $smallest
```

```
#!/bin/bash
smallest=10000
for i in 5 8 19 8 7 3
  do
      if [ $i -lt
      $smallest ]
        then
            smallest=$i
      fi
done
echo $smallest
```

The While Loop

- While loops repeat statements as long as the next Unix command is successful.
- Works similar to the while loop in C.

```
#! /bin/bash
i=1
sum=0
while [ $i -le 100 ]
do
sum=`expr $sum + $i`
i=`expr $i + 1`
done
echo The sum is $sum.
```

The Until Loop

- Until loops repeat statements until the next Unix command is successful.
- Works similar to the do-while loop in C.

```
#! /bin/bash
x=1
until [ $x -gt 3 ]
do
        echo x = $x
        x=`expr $x + 1`
done
```

Command Line Arguments

Reading input in shell programs

Command Line

- Parameters to any program.
  - E.g:
  - $ ls –t foo
  - ‘-t’ and foo are parameters to the program ls.
- The command line for ls now consists of three parameters: ls, -t and foo
- Shell script arguments are “numbered” from left to right
  - $1 - first argument after command.
  - $2 - second argument after command.
  - ... up to $9.
  - They are called “positional parameters”.
    - Their position in the command line determines their value.

Command Line

- Ex: find out if string appears in file.
  - Run command as: $ mystr string file

```
bash-3.2$ cat myscript
#! /bin/bash
grep $1 $2
bash-3.2$ ./myscript setenv testfile
setenv TERM xterm
setenv EDITOR /usr/ucb/vi
setenv MAIL /usr/spool/mail/$USER
setenv MAILER /usr/ucb/mail
setenv PAGER more
setenv PRINTER hp
bash-3.2$
```

NOTE: $1 has value setenv and

$2 has value testfile, part of a .login file

Command Line

- Other variables related to arguments:
  - $0 ! Name of the command running.
  - $* ! All the arguments (even if there are more than 9).
  - $# ! The number of arguments.

```
$ cat cmd_line
#! /bin/bash
echo "$0 is the name of the command"
echo "$* is the list of arguments"
echo "$# is the total number of arguments”
```

Command Line

- Example Output:

```
bash-3.2$ ./cmd_line
./cmd_line is the name of the command
is the list of arguments
0 is the total number of arguments

bash-3.2$ ./cmd_line 1 2 3 4 5 6 7
./cmd_line is the name of the command
1 2 3 4 5 6 7 is the list of arguments
7 is the total number of arguments
bash-3.2$
```

More on Bash Variables

- There are three basic types of variables in a shell script:
  - Positional variables ...
    - $1, $2, $3, ..., $9
  - Keyword variables ...
    - Like $PATH, $HOME, and anything else we may define.
  - Special variables
    - $! - return process id of last background process to finish
    - $? - return status of last foreground process to finish
    - $$ - the process id of the current shell
    - There are others you can find out about with man sh

Reading Input

- Done using the read command.
  - Reads one line of input and assigns it to variables given as arguments.
    - Data type of variable does not matter, as shell has no concept of data types.
- Syntax:
  - read var1, var2, var3 ….
  - Reads a line of input from standard input.
  - Assigns first word to var1, second word to var2, ...
  - The last variable gets any excess words on the line.

Notes on Read

- Example:

```
bash-3.2$ read var1 var2 var3
hello world again with some extra on the
end
bash-3.2$ echo $var1
hello
bash-3.2$ echo $var2
world
bash-3.2$ echo $var3
again with some extra on the end
bash-3.2$
```

NOTE: var3 has the rest of the string

The Case Statement

- Falls into the category of conditional statements.
- Allows the user to branch depending on the outcome of a string.
- Different from C, where the outcome could only be an integral value (char, int).

```
Syntax:
    case string in
        pattern1)
            command_set_1
            ;;
        pattern2)
            command_set_2
            ;;
        ...
    esac
```

Example

```
#!/bin/bash
echo -n 'Choose option [1-2]
> '
read reply
case $reply in
    "1")
        echo “the choice was 1”
        ;;
    "2")
        echo “the choice was 2”
        ;;
    *)
        echo Illegal choice!
        ;;
esac
```

Provide a default case when no other cases are matched.

Notes on Case

- Possible to combine two outcomes into one.
  - Using the logical OR in shell.

```
case $reply in
        “1” | “2”)
                echo “The choice is either 1 or 2”
                ;;
        *)
                echo “wrong choice”
                ;;
esac
```

Notes on Case

- The outcome is always checked as a string.
- The ';;' are necessary to tell the shell that this option to the case is over.
- Every case statement must be terminated with an esac

Reference Books

- GNU/Linux Command–Line Tools Summary
- Very good summary of commands
- 2 BASH books, beginners and advanced
- Both on Moodle under Week 7