UNIX shell scripting

EECS 2031

Summer 2014

Przemyslaw Pawluk

June 17, 2014

What we will discuss today

Introduction

Control Structures

User Input

Homework

Table of Contents

What is Shell

- A program that interprets your requests to run other programs
- Most common Unix shells:
  - Bourne shell (sh)
  - C shell (csh - tcsh)
  - Korn shell (ksh)
  - Bourne-again shell (bash)
- In this course we focus on Bourne shell (sh)

The Bourne Shell

- A high level programming language
- Processes groups of commands stored in files called scripts
- Includes
  - variables
  - control structures
  - processes
  - signals

Executable Files

- Contain one or more shell commands.
- These files can be made executable.
- # indicates a comment
- Except on line 1 when followed by an !

```
#!/bin/sh
echo 'Hello World!'
```

Executable Files

```
% cat welcome
#!/bin/sh
echo Hello World!
% welcome
welcome: Permission denied.
% chmod 744 welcome
% ls -l welcome
-rwxr--r-- 1 bil faculty 30 Sep 19 11:02 welcome
% welcome
Hello World!
% welcome > greet_them
% cat greet_them
Hello World!
```

Executable Files

- ?If the file is not executable, use `sh` followed by the file name to run the script.

```
% chmod 644 welcome
% ls -l welcome
-rw-r--r-- 1 bil faculty 30 Sep 19 10:49 welcome
% sh welcome
Hello World!
```

Variables

- Standard UNIX variables
  - Consist of shell variables and environment variables.
  - Used to tailor the operating environment to suit your needs.
  - Examples: TERM, HOME, PATH
  - To display your environment variables, type set.
- User variables: variables you create yourself.
- Positional parameters
- Store the values of command-line arguments.

User Variables

- Syntax: `name=value`
- No space around the equal sign!
- All shell variables store strings (no numeric values).
- Variable name: combinations of letters, numbers, and underscore character (_) that do not start with a number.
- Avoid existing commands and environment variables.
- Shell stores and remembers these variables and supplies value on demand.

User Variables

- To use a variable: `$varname`
- Operator $ tells the shell to substitute the value of the variable name.

```
% cat ma
#!/bin/sh
dir=/usr/include/
echo $dir
echo dir
ls $dir | grep 'ma'
```

Output

- What if I want to display the following?
  `$dir`
- Two ways to prevent variable substitution:
  `echo '$dir'`
  `echo \$dir`
- Note:
  `echo "$dir"` does the same as
  `echo $dir`

User Variables and Quotes

- If value contains no space, no need to use quotes: `dir=/usr/include/`
- Unless you want to protect the literal $

```
% cat quotes
#!/bin/sh
# Test values with quotes
myvar1=$100
myvar2='$100'
echo The price is $myvar1
echo The price is $myvar2
```

Variables and Quotes

If value contains one or more spaces:

- Use single quotes for NO interpretation of meta-characters (protect the literal)
- Use double quotes for interpretation of meta-characters

Example

```
% cat quotes2
#!/bin/sh
myvar=‘whoami‘
squotes=’Today is ‘date‘, $myvar.’
dquotes="Today is ‘date‘, $myvar."
echo $squotes
echo $dquotes
```

Table of Contents

Control Structures

- `if then else`
- `for`
- `while`
- `case (which)`
- `until`

if and test commands

Syntax

```
if condition
then
 command(s)
elif condition_2
then
 command(s)
else
 command(s)
fi
```

test command

- -e arg True if arg exists in the file system
- -d arg True if arg is a directory
- -f arg True if arg is an ordinary file
- -r arg True if arg is readable
- -w arg True if arg is writable
- -x arg True if arg is executable
- -s arg True if size of arg is greater than 0
- ! d arg True if arg is not a directory 1

test – Numeric tests

- n1 –eq n2 $n1 == n2$
- n1 –ge n2 $n1 >= n2$
- n1 –gt n2 $n1 > n2$
- n1 –le n2 $n1 <= n2$
- n1 –ne n2 $n1! = n2$
- n1 –lt n2 $n1 < n2$

case Statement

```
case variable in
pattern1) command(s);;
pattern2) command(s);;
. . .
patternN) command(s);;
*) command(s);; # all other cases
esac
```

for Loop

```
for variable in list
do
  command(s)
done
```

- variable is a user-defined variable.
- list is a sequence of strings separated by spaces

Example

```
% cat fingr
#!/bin/sh
for name in $*
do
 finger $name
done
```

Recall that $* stands for all command line arguments the user enters.

Arithmetic Operations Using expr

It's not for numerical work

but you can use `expr` utility may be used for simple arithmetic operations on integers

- `expr` is not a shell command but rather a UNIX utility
- To use expr in a shell script, enclose the expression with backquotes.

Example

```
#!/bin/sh
sum=‘expr $1 + $2‘
echo $sum
l? Note: spaces are
```

while Loop

```
while condition
 do
    command(s)
done
```

- Command test is often used in condition.
- Execute command(s)when condition is met.

until Loop

```
until condition
do
  command(s)
done
```

- Command test is often used in condition.
- Exit loop when condition is met.

break and continue

- Interrupt loops (for, while, until)
- `break` transfers control immediately to the statement after the nearest done statement
  - terminates execution of the current loop
- `continue` transfers control immediately to the nearest `done` statement
  - brings execution back to the top of the loop

Shell Functions

- Similar to shell scripts.
- Stored in shell where it is defined (instead of in a file).
- Executed within `sh`
  - no child process spawned
- Syntax:

```
function_name()
{
    commands
}
```

- Allows structured shell scripts

Table of Contents

Reading User Input

- Reads from standard input.
- Stores what is read in user variable.
- Waits for the user to enter something followed by `<RETURN>`
- Syntax:

```
read varname # no dollar sign $
```

- To use the input:

```
echo $varname
```

Readin User Input

- More than one variable may be specified.
- Each word will be stored in separate variable.
- If not enough variables for words, the last variable stores the rest of the line.

Command Line Arguments

- Command line arguments stored in variables are called positional parameters.
- These parameters are named $1 through $9.
- Command itself is in parameter $0.
- In diagram format:

command	arg1	arg2	arg3	arg4	arg5	arg6	arg7	arg8	arg9
$0	$1	$2	$3	$4	$5	$6	$7	$8	$9

Command Line Arguments

- $# represents the number of command line arguments
- $* represents all the command line arguments
- $@ represents all the command line arguments

```
% cat check_args
#!/bin/sh
echo There are $# arguments.
echo All the arguments are: $*
# or echo All the arguments are: $@

% check_args Mary Tom Amy Tony
There are 4 arguments.
All the arguments are: Mary Tom Amy Tony
```

What if there is more than 9 arguments?

- If the number of arguments is more than 9? How to access the 10th, 11th, etc.?
- Use shift operator.

shift Operator

- shift promotes each argument one position to the left
- Allows access to arguments beyond $9.
- Operates as a conveyor belt.
  - Shifts contents of $2 into $1
  - Shifts contents of $3 into $2
  - Shifts contents of $4 into $3 etc.
- Eliminates argument that used to be in $1
- After a shift, the argument count stored in $# is automatically decreased by one.

Environment and Shell Variables

- Standard UNIX variables are divided into 2 categories: *shell variables* and *environment variables*.
- Shell variables: apply only to the current instance of the shell; used to set short-term working conditions.
  - displayed using set command.
- Environment variables: set at login and are valid for the duration of the session.
  - displayed using env command.
- By convention, environment variables have UPPER CASE and shell variables have lower case names.

Environment and Shell Variables

- In general, environment and shell variables that have "the same" name (apart from the case) are distinct and independent, except for possibly having the same initial values.
- Exceptions:
  - When home, user and term are changed, HOME, USER and TERM receive the same values.
  - But changing HOME, USER or TERM does not affect home, user or term.
  - Changing PATH causes path to be changed and vice versa.

Variable path

PATH and path specify directories to search for commands and programs

```
cd # current dir is home dir
funcex # this fails because funcex

# is in www/2031/Lecture3
set path=($path www/2031/Lecture3)

funcex # successful
```

To add a path permanently, add the line to your `.cshrc` file after the list of other commands.

```
set path=($path .)
```

Process running

- Each running program on a UNIX system is called a process.
- Processes are identified by a number (process id or PID).
- Each process has a unique PID.
- There are usually several processes running concurrently in a UNIX system.

Table of Contents

Homework

Review material for the mid-term exam that will be handled next week!

- data types and sizes
- expressions types and values
- declaring variables of a given types
- keywords such as `static`, `extern`, `const`
- analyzing C programs (including finding errors)
- writing C code based on the specification