CSCI 330
The UNIX System

Introduction to Shell Programming

Introduction to Shell Scripts

- shell programming is one of the most powerful features on any UNIX system
- large portion on UNIX administration and house keeping is done via shell scripts
- if you cannot find an existing utility to accomplish a task, you can build one using a shell script
- Shell scripts can do what can be done on the command line

Shell Scripts

- A shell program contains high-level programming language features:
  - Variables for storing data
  - Decision-making control (e.g. if and case statements)
  - Looping abilities (e.g. for and while loops)
  - Function calls for modularity
- A shell program can also contain:
  - any UNIX command
  - file manipulation: cp, mv, ls, cd, …
  - utilities: grep, sed, awk, …
- Comments: lines starting with '#'

Shell Script: the basics

- 1. line (shebang line) for bash shell script:

```
    #!/bin/bash
   #!/bin/sh
```

- to run:
  - make executable: `% chmod +x script`
  - invoke via: `% ./script`

bash Shell Programming Features

- Variables
- Input/output
  - command line parameters
  - prompting user
- Decision
  - if-then-else
  - case
- Repetition
  - do-while, repeat-until
  - for, select
- Functions
- Traps

User-defined shell variables

Syntax:

```
varname=value
```

Note: no spaces

Example:

```
rate=moderate
echo “Rate today is: $rate”
```

- Use quotes if the value of a variable contains white spaces (double quotes preferred)

Example:

```
name=“Thomas William Flowers”
```

Output via echo command

- Simplest form of writing to standard output

<u>Syntax:</u> echo [-ne] argument[s]

-n suppresses trailing newline

-e enables escape sequences:

\t	horizontal tab
\b	backspace
\a	alert
\n	newline

Examples: shell scripts with output

```
#!/bin/bash
echo "You are running these processes:"
ps
```

```
#!/bin/bash
echo -ne "Dear $USER:\nWhat's up this month:"
cal
```

Command line arguments

- Use arguments to modify script behavior
- command line arguments become positional parameters to shell script
- positional parameters are numbered variables: $1, $2, $3 …

Command line arguments

	Meaning
$1	first parameter
$2	second parameter
${10}	10th parameter { } prevents “$1” misunderstanding
$0	name of the script
$*	all positional parameters
$#	the number of arguments

Example: Command Line Arguments

```
#!/bin/bash
# Usage: greetings name1 name2

echo $0 to you $1 $2
echo Today is `date`
echo Good Bye $1
```

Script example

- Use command line argument as input for command

```
#!/bin/bash
# counts characters in command argument
echo -n "$1" | wc -c
```

Arithmetic expressions

Syntax:

```
$((expression))
```

- can be used for simple arithmetic:

```
count=1
echo $((count+20))
echo $((count++))
```

Array variables

<u>Syntax:</u>

```
varname=(list of words)
```

- accessed via index:

`${varname[index]}`	
`${varname[0]}`	first word in array
`${varname[*]}`	all words in array
`${#varname[*]}`	number of words

Using array variables

Examples:

```
% ml=(mary ann bruce linda dara)
% echo $ml
mary
% echo ${ml[*]}
mary ann bruce linda dara
% echo ${ml[2]}
bruce
% ml[2]=john
% echo ${ml[*]}
mary ann john linda dara
```

Variables commands

- To delete both local and environment variables

```
    unset varname
```

- To prohibit change

```
    readonly varname
```

- list all shell variables (including exported)

```
    set
```

variable manipulation - substring

- use portion of a variable's value via:

  `${name:offset:length}`

  - name – the name of the variable
  - offset – beginning position of the value
  - length – the number of positions of the value

<u>Example:</u>

```
% SSN="123456789"

% password=${SSN:5:4}

% echo $password

% 6789                    (Why?)
```

Special variable uses

- **`${#variable}`**

  number of characters in variable’s value

- **`${variable:-value}`**

  if variable is undefined use “value” instead

- **`${variable:=value}`**

  if variable is undefined use “value” instead, and set variable’s value

- **`${varname:?message}`**

  if variable is undefined display error “message”

Output

- common commands
  - echo
  - printf

Syntax: echo [-ne] arguments

-n suppresses trailing newline

-e enables escape sequences:

\t	horizontal tab
\b	backspace
\a	alert
\n	newline

Output: printf command

Syntax: printf format [arguments]

- writes formatted arguments to standard output under the control of "format"

- format string may contain:
  - plain characters: printed to output
  - escape characters: e.g. \t, \n, \a …
  - format specifiers: prints next successive argument

printf format specifiers

%d	number	(decimal integer)	
	also:	%10d	10 characters wide
		%-10d	left justified
%s	string		
	also:	%20s	20 characters wide
		%-20s	left justified

Examples: printf

```
% printf “random number”

% printf “random number\n”

% printf “random number: %d” $RANDOM

% printf “random number: %10d\n” $RANDOM

% printf “%d for %s\n” $RANDOM $USER
```

User input

- shell allows to prompt for user input

<u>Syntax:</u>

```
read [–p “prompt”] varname [more vars]
```

- words entered by user are assigned to **`varname`** and “**`more vars`**”
- last variable gets rest of input line

Example: Accepting User Input

```
#!/bin/bash
read -p "enter your name: " first last

echo "First name: $first"
echo "Last name: $last"
```

Exit Command

- Terminates the current shell, the running script

- Syntax   exit[status]
  - Default exit status is 0 (contrary to C programming)
- `% exit`
- `% exit 1`
- `% exit -1`

Exit Status

- Also called: return status
- Predefined variable “?” holds exit status of last command
- 0 indicates success, all else is failure

```
% ls > /tmp/out

% echo $?

% grep -q “root” boot.log

% echo $?
```

bash Control Structures

- if-then-else
- case
- loops
  - for
  - while
  - until
  - select

Conditional Execution

- Operators || and && allow conditional execution
  - Lazy evaluation, shortcut execution
- **`cmd1 && cmd2`**
  - cmd2 executed if cmd1 succeeds
- **`cmd1 || cmd2`**
  - cmd2 executed if cmd1 fails
- Perform boolean “or” “and” on exit status

Conditional examples

- `% grep $USER /etc/passwd && echo “$USER found”`
  - If left is true then do the right

- `%grep student /etc/group || echo “no student group”`
  - If left is not true then do the right

Summary

- Shell scripts can do what can be done on command line
- Shell scripts simplify recurring tasks.