Variables, Quotes, Passing Arguments

Outline

- Command Files
- Variables
- Built-in Integer Arithmetic
- Quotes and Backslash
- Command Substitution
- Passing Arguments

Command Files (1)

A shell program can be typed directly at the terminal, as in

$who | wc -l

or it can be first typed into a file and then the file can be executed by the shell.

To accomplish the latter:

1. Open a text editor, for example type **vi nu.sh** in the OS prompt. This will open the editor vi. Then, type **i** to insert text.
2. Type the command: **who | wc -l**
3. Save the file nu.sh. To save in vi press **Esc** then type **:wq** and press **Enter**
4. Make the program executable: **chmod +x nu.sh**
5. Run the file: **./nu.sh**

Command Files (2)

Important Note:

We will be using the bash shell on uisacad5 to run the shell programs we write. This should be the default shell on the server.

$ echo $0	***check your current shell***
$ /bin/bash	***change to bash shell***
$ echo $SHELL	***check the login shell***

Outline

- Command Files
- Variables
- Built-in Integer Arithmetic
- Quotes and Backslash
- Command Substitution
- Passing Arguments

Variables

• A variable is a location in memory where values can be stored.

• To store a value inside a shell variable, you simply write the name of the variable, followed immediately by the equal sign =, followed immediately by the value.

Examples:

variable_name=value
count=1

There are two variable classifications:

1. User-Defined Variables are defined by users.
2. Predefined Variables are part of a user's shell environment. Ex. systems variables.

Two important points:

1. Spaces are not permitted on either side of the equal sign.

2. The shell has no concept of data types. The value assigned is always interpreted as string characters.

Storing Variables

All three shells provide means to store values in variables. The Korn and Bash shells are the same. C shell is different as shown in Table 3.1.

Action	Korn and Bash	C Shell
Assignment	variable=value	set variable = value
Reference	$variable	$variable

Table 3.1 Variable Shell Commands

Displaying Variables

Displaying the values of variables:
echo $variable_name

Examples:

$ count=99 *assign character 1 to count*

$ my_prog=/home/kzepp2/prog *assign /home/kzepp2 my_prog*

$ echo $count *display content of count*

99

$ echo $my_prog $count *display contents of my_prog and count*

/home/kzepp2/prog 99

$ echo There are $count bottles on the wall.

There are 99 bottles on the wall.

Filename Substitution and Variables

Given:

$ ls

addresses
intro
lotsaspaces
names
nu
numbers
phonebook
stat

$ x=*

$ echo $x

addresses intro lotsaspaces names nu numbers phonebook stat

Question:

Was the list of files stored into the variable x when x=* was executed?

Filename Substitution and Variables (2)

Answer: When **echo $x** was executed, the following steps were followed:

1. The shell scanned the line, substituting * as the value of x.
2. The shell rescanned the line, encountered the *, and then substituted the names of all files in the current directory.
3. The shell initiated execution of echo, passing it the file list as arguments.

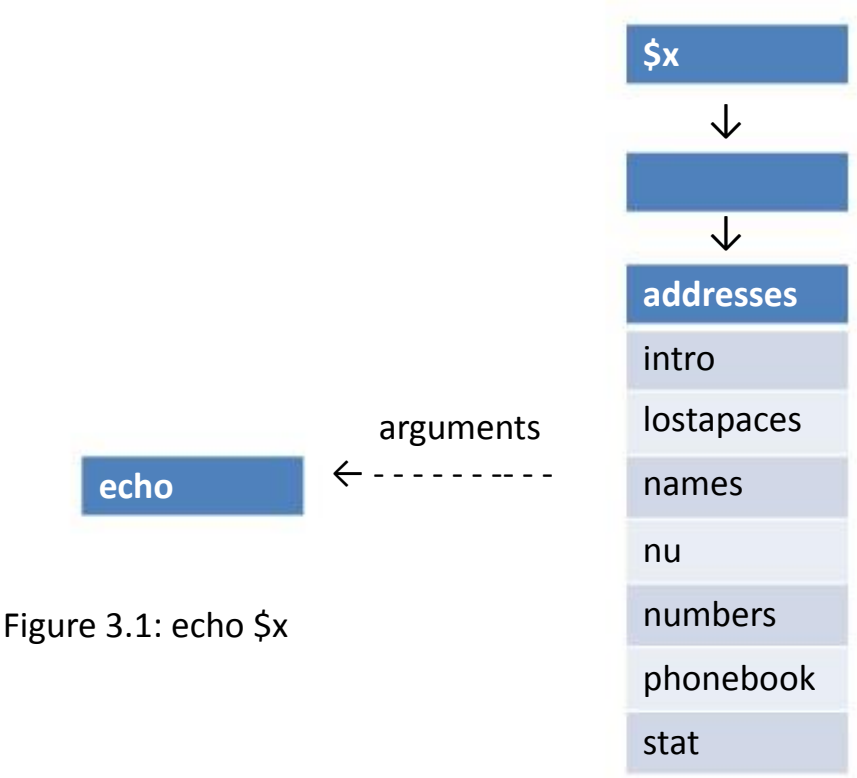

Figure 3.1: echo $x

Predefined Variables

Table 3.2 Predefined Variables

Korn and Bash	C Shell	Explanation
CDPATH	cdpath	Contains the search path for cd command when the directory argument is a relative pathname
EDITOR	EDITOR	Pathname of the command-line editor
ENV		Pathname of the environment file.
HOME	home (HOME)	Pathname for the home directory.
PATH	path (PATH)	Search path for commands.
PS1	prompt	Primary prompt, such as $ and %.
SHELL	shell (SHELL)	Pathname of the login shell.
TERM	term (TERM)	Terminal type.
TMOUT	autologout	Defines idle time, in seconds, before shell automatically logs you off.
VISUAL	VISUAL	Pathname of the editor for command-line editing.

Outline

- Command Files
- Variables
- Built-in Integer Arithmetic
- Quotes and Backslash
- Command Substitution
- Passing Arguments

Built-in Integer Arithmetic

An arithmetic expression is the result of an arithmetic operation. In arithmetic operation, it matches the value when it is non-zero.

Format:
$ ((*expression*))

where *expression* is an arithmetic expression using shell variables and operations. It consists of operators, numbers and variables.

Examples:

1. **echo $((i+1))** — *this adds one to the value in the shell variable **i** and prints the result.*

2. **echo $((a=a+1))** — *equivalent to a = 0 + 1, and prints 1*

3. **i=$((i*5))** — *multiply the variable **i** by 5 and assign the result back to **i***

Arithmetic Expressions

Table 3.3 Arithmetic Expression Examples

Operator	Example	Explanation
* / % ^	a^2	Variable a is raised to power 2 (a^2).
++	++a a++	Adds 1 to a.
--	--a a--	Subtracts 1 from a.
+ -	a + b a-b	Adds or subtracts two values.
+	+a	Unary plus: Value is unchanged.
-	-a	Unary minus: Value is complemented.
=	a = 0	a is assigned the value 0.
*=	x *= y	The equivalent of x =x *y; x is assigned the product of x * y.
/=	x /= y	The equivalent of x =x /y; x is assigned the quotient of x / y.
%=	x %= y	The equivalent of x =x %y; where % is the modulo operator; x is assigned the modulus of x / y.
+=	x += 5	The equivalent of x =x +5; x is assigned the sum of x and 5.
-=	x -= 5	The equivalent of x =x -5; x is assigned the difference of (x -5).

Outline

- Command Files
- Variables
- Built-in Integer Arithmetic
- Quotes and Backslash
- Command Substitution
- Passing Arguments

Single Quote (')

Single Quotes are used to keep characters separated by whitespace characters together. Any enclosed characters are treated as literal characters.

Examples:

1. **$echo one two three four**
   one two three four
   extra spaces are removed

2. **$echo 'one two three four '**
   one two three four
   extra spaces are preserved

3. **$ grep 'Susan Goldberg' phonebook**
   Susan Goldberg 201-555-7776
   takes Susan Goldberg as one argument

4. **$ echo 'One two "three" four'**
   One two "three" four
   Quotes inside quotes
   when it is necessary to use a double quote around "three", use single quote as the outside quote

Double Quote (")

Double quotes work similarly to single quotes, except that they're not as restrictive. Whereas the single quotes tell the shell to ignore *all* enclosed characters, double quotes say to ignore *most* enclosed characters except for the following:

- Dollar signs
- Back quotes
- Backslashes

Example:

$ x=* *assign variable x with **

$ echo $x *display $x (no quote)*

data intro lotsaspaces names phonebook README.txt text.out

**$ echo '$x'** *display '$x' (single quote) = as is*

$x

**$ echo "$x"** *display "$x" (double quote) = content of x*

*

$

The Backslash (\)

The backslash is equivalent to placing single quotes around a single character, with a few minor exceptions. It removes the special meaning of the character that follows. The backslash quotes the single character that immediately follows it.

Format:
c

where *c* is the character you want to quote

The Backslash (\) - Example

Examples:

$ echo >

bash: syntax error near unexpected token `newline‘

$ echo \>

>

$ x=*

$ echo \$x

$x

**$ echo **

\

$ echo '\'

\

$

Outline

- Command Files
- Variables
- Built-in Integer Arithmetic
- Quotes and Backslash
- Command Substitution
- Passing Arguments

Command Substitution (1)

Command substitution refers to the shell's capability to insert the standard output of a command at any point in a command line.
Two ways to perform command substitution:

1. Use of back quotes
2. $(…) construct.

The Back Quote:

Its purpose is to tell the shell to execute the enclosed command and to insert the standard output from the command at that point on the command line.

Format:

`command`

Example:

$ echo The date and time is: `date`

The date and time is: Sat Aug 30 23:58:53 CDT 2008

Command Substitution (2)

The $(…) Construct - preferred method. Supported in newer UNIX or Linux

Example No. 1

$ echo The date and time is: $(date)

The date and time is: Sun Aug 31 00:07:13 CDT 2008

Example No. 2

$ echo '$(who | wc -l) tells how many users are logged in'

$(who | wc -l) tells how many users are logged in

$ echo "$(who | wc -l) tells how many users are logged in"

3 tells how many users are logged in

$

The expr Command

expr evaluates an expression given to it on the command line.

Examples:

1. **$ expr 1 + 2** *There must be space in between the operator and operand.*

   3

2. **$ expr 1 * 2** *Use backspace for multiplication*

   2

3. **$ i=8** *Assign i with 8*

   $ expr $i / 2 *divide i with 2*

   4 *Gives 4*

4. **$ i=8**

   $ i=$(expr $i + 1) *Add 1 to i*

   $ echo $i

   9

Outline

- Command Files
- Variables
- Built-in Integer Arithmetic
- Quotes and Backslash
- Command Substitution
- Passing Arguments

Passing Arguments

Shell programs can take arguments typed on the command line as input.

Example:

$ cat ison *display content of ison*

```
who | grep $1
```

$who *see who is logged in*

```
root     pts/0       Aug 29 09:24
rsalv1   pts/1       Sep  3 19:27
```

$ ./ison root *run the program, ison*

```
root     pts/0       Aug 29 09:24 (10.100.150.13)
```

The $# Variable

$# gets the number of arguments that were typed on the command line.

Example:

$ ls args

-rwxr-xr-x 1 kzepp2 users 69 Sep 3 20:29 args

$ cat args

echo $# arguments passed

echo arg 1 = :$1: arg 2 = :$2: arg 3 = :$3:

$

$ ./args a b c d

4 arguments passed

arg 1 = :a: arg 2 = :b: arg 3 = :c:

$

Note:

Please create and run *args* yourself and enter different arguments and study the output for each run with different arguments passed. Do not forget to make the file executable.

The $* Variable

The special variable **$*** references all the arguments passed to the program. This is often useful in programs that take an indeterminate or variable number of arguments.

Example on next slide

The $* Variable

Examples:

$ cat args2

```
echo $# arguments passed
echo they are :$*:
```

$ ./args2 a b c

```
3 arguments passed
they are :a b c:
```

$ ./args2 a b c d

```
4 arguments passed
they are :a b c d:
```

$ ./args2 one two

```
2 arguments passed
they are :one two:
```

$ ./args2

```
0 arguments passed
they are ::
```

$ ./args2 *

```
8 arguments passed
they are :args args2 book chap2 ison nu.sh run sys.caps:
$
```

The shift Command (1)

The **shift** command allows you to effectively left shift your positional paramters. If you execute the command shift, whatever was previously stored inside $2 will be assigned to $1, whatever was previously stored in $3 will be assigned to $2, and so on. The old value of $1 will be irretrievably lost.

Example:

```
$cat  tshift                    check what is inside tshift
echo $# $*
shift
echo $# $*
shift
echo $# $*
shift
echo $# $*
shift
echo $# $*
shift
echo $# $*
shift
echo $# $*
shift
```

The shift Command (2)

(continuation …)

$./tshift a b c d e *run the program*

5 a b c d e

4 b c d e

3 c d e

2 d e

1 e

0

$

References

1. Unix Shell Programming, Kochan and Wood, 3rd Ed., Chapter 5, 6, and7
2. Unix and Shell Programming, Foruozan and Gilberg, 2003, Chapter 5, 12