# UNX510/DPS918 - Unix BASH Shell Scripting

## Lecture 12 - Variable Expansion; Shell Expansion; Shell Startup; alias

### Variable Expansion

* $var - "$" before a variable name will retrieve it's value, also called variable substitution
* ${var} - brace brackets can be used to delineate the variable name from following characters
* ${!var} - will use value of "var" as the variable whose value will be retrieved - this is called indirect expansion:
* ==> set one two three four five
* ==> num=3
* ==> echo ${num}
* 3
* ==> echo ${!num}
* three
* ==> varName=num
* ==> echo ${varName}
* num
* ==> echo ${!varName}
* 3
* ==> \_
* note that this can also be done with eval :
* ==> set one two three four five
* ==> num=3
* ==> echo ${num}
* 3
* ==> echo $${num}
* 7395{num}
* ==> echo \$${num}
* $3
* ==> eval echo \$${num}
* three
* ==> varName=num
* ==> echo ${varName}
* num
* ==> eval echo \$${varName}
* 3
* ==> \_
* note that combining eval with indirect expansion gives a double indirection:
* ==> echo ${varName}
* num
* ==> echo ${!varName}
* 3
* ==> eval echo \$${!varName}
* three
* ==> \_
* indirect expansion can be used to get the value of the last positional parameter:
* ==> set one two three four five
* ==> args=$#
* ==> lastarg=${!args}
* ==> echo $lastarg
* five
* ==> echo ${!#}
* five
* ==> \_
* ${var:-default} - expands to "default" value if "var" is null or unset, "var" is unchanged
* ${var:+default} - expands to "default" value if "var" is set, "var" is unchanged
* ${var:?errmsg} - sends "errmsg" to stderr if "var" is null or unset, "var" is unchanged, if the shell is non-interactive it will exit
* ${var:=default} - sets "var" to "default" value if "var" is null or unset
* here are the results when the variable is not set:
* ==> echo $var
* ==> echo ${var:-"this default value will be displayed"}
* this default value will be displayed
* ==> echo $var
* ==> echo ${var:+"this default value will not be displayed"}
* ==> echo ${var:?}
* -bash: var: parameter null or not set
* ==> echo ${var:?"this error message will be sent to stderr"}
* -bash: var: this error message will be sent to stderr
* ==> echo $?
* 1
* ==> echo ${var:="var now has this value"}
* var now has this value
* ==> echo $var
* var now has this value
* ==> \_
* without the colon ":", the variable would be checked for existence only, the value would not be checked for null
* here are the results with the variable set:
* ==> echo $var
* var now has this value
* ==> echo ${var:-"this default value will not be displayed"}
* var now has this value
* ==> echo ${var:+"this default value will be displayed"}
* this default value will be displayed
* ==> echo ${var:?}
* var now has this value
* ==> echo ${var:?"this error message will not be sent to stderr"}
* var now has this value
* ==> echo $?
* 0
* ==> echo ${var:="this default value will not be displayed"}
* var now has this value
* ==> echo $var
* var now has this value
* ==> \_
* ${var:offset:length} - substring substitution, with an optional length
* positive offset (from beginning of string) for first character is 0
* negative offset (from end of string) for last character is -1
* positive length means length of substring
* negative length is offset from end of string
* here are some examples of substring substitution:
* ==> var1=abcdefghijklmnopqrstuvwxyz
* ==> echo ${var1:9}
* jklmnopqrstuvwxyz
* ==> echo ${var1:9:5}
* jklmn
* ==> echo ${var1:9:-5}
* jklmnopqrstu
* ==> echo ${var1:-9} # misinterpreted as a ":-" operator
* abcdefghijklmnopqrstuvwxyz
* ==> echo ${var1: -9}
* rstuvwxyz
* ==> echo ${var1: -9:4}
* rstu
* ==> echo ${var1: -9:-4}
* rstuv
* ==> set abcdefghijklmnopqrstuvwxyz
* ==> echo ${1: -9:-4}
* rstuv
* ==> num1=14; num2=7
* ==> echo ${1: $((num1-5)): $((num2-2))}
* jklmn
* ==> \_
* ${#var} - gives the length of "$var"
* ${!var\*} - displays all variable names beginning with "var" as a single string
* ${!var@} - displays all variable names beginning with "var" as separate strings
* here are some examples:
* ==> var1=lion
* ==> var2=tiger
* ==> var3="bear, oh my"
* ==> echo ${#var3}
* 11
* ==> echo ${!var\*}
* var1 var2 var3
* ==> \_
* ${var%pattern} - removes the shortest part of "$var" that matches "pattern", from the end
* ${var%%pattern} - removes the longest part of "$var" that matches "pattern", from the end
* ${var#pattern} - removes the shortest part of "$var" that matches "pattern", from the beginning
* ${var##pattern} - removes the longest part of "$var" that matches "pattern", from the beginning
* if variable is "\*" or "@", the action is applied to each positional parameter
* useful for string manipulation, for example directory processing:
* ==> x=$PWD
* ==> echo $x
* /home/lczegel/test/dir1
* ==> echo ${#x}
* 23
* ==> echo ${x%/\*}
* /home/lczegel/test
* ==> echo ${x%%/\*}
* ==> echo ${x#\*/}
* home/lczegel/test/dir1
* ==> echo ${x##\*/}
* dir1
* ==> \_
* here is an example of setting the primary prompt:
* ==> PS1='$PWD: '
* /home/lczegel/test/dir1: cd
* /home/lczegel: cd -
* /home/lczegel/test/dir1
* /home/lczegel/test/dir1: PS1='${PWD#~/}: '
* test/dir1: cd
* /home/lczegel: cd -
* /home/lczegel/test/dir1
* test/dir1: \_
* ${var/pattern/string} - replaces the longest part of "$var" that matches "pattern", with "string"
* defaults to replacing the first match
  + if pattern begins with /, all matches will be replaced
  + if pattern begins with #, the match must be at the beginning of "$var"
  + if pattern begins with %, the match must be at the end of "$var"
* if variable is "\*" or "@", the action is applied to each positional parameter
* useful for simple string manipulation:
* ==> var1="My SCHOOL is seneca college"
* ==> echo ${var1/seneca/the best}
* My SCHOOL is the best college
* ==> echo ${var1/e/a}
* My SCHOOL is saneca college
* ==> echo ${var1//e/a}
* My SCHOOL is sanaca collaga
* dir1
* ==> \_
* ${var^pattern} - converts the first character of substrings that match "pattern" to uppercase
* ${var^^pattern} - converts all characters of substrings that match "pattern" to uppercase
* ${var,pattern} - converts the first character of substrings that match "pattern" to lowercase
* ${var,,pattern} - converts all characters of substrings that match "pattern" to lowercase
* if pattern is missing, default is "?", matching all characters in "$var"
* if variable is "\*" or "@", the action is applied to each positional parameter
* useful for simple string manipulation:
* ==> echo $var1
* My SCHOOL is seneca college
* ==> echo ${var1^}
* My SCHOOL is seneca college
* ==> echo ${var1^^}
* MY SCHOOL IS SENECA COLLEGE
* ==> echo ${var1,}
* my SCHOOL is seneca college
* ==> echo ${var1,,}
* my school is seneca college
* ==> echo ${var1^^e}
* My SCHOOL is sEnEca collEgE
* ==> echo ${var1,,O}
* My SCHooL is seneca college
* ==> \_

### Shell Expansion

* the shell performs eight kinds of expansion, in the following order:
  + brace expansion
  + tilde expansion
  + variable expansion, command substitution, arithmetic expansion
  + process substitution
  + word splitting
  + pathname expansion

### Brace Expansion

* can generate sequences of character strings
* similar idea to pathname expansion, without the need to match existing filenames
* can use a comma-separated list, or a sequence
* will not be expanded within quotes
* here are some examples of brace expansion:
* ==> echo {1,2,3,5,8,13,21}
* 1 2 3 5 8 13 21
* ==> echo {1..10}
* 1 2 3 4 5 6 7 8 9 10
* ==> echo {01..10}
* 01 02 03 04 05 06 07 08 09 10
* ==> echo {1..10..2}
* 1 3 5 7 9
* ==> echo {10..1}
* 10 9 8 7 6 5 4 3 2 1
* ==> echo {10..01..-2}
* 10 08 06 04 02
* ==> echo {a..t..2}
* a c e g i k m o q s
* ==> \_
* here are some examples using brace expansion to generate filenames:
* ==> touch file{1..10..2}.txt
* ==> ls
* file1.txt file3.txt file5.txt file7.txt file9.txt
* ==> rm file{0..10..3}.\*
* rm: cannot remove `file0.\*': No such file or directory
* rm: cannot remove `file6.\*': No such file or directory
* ==> ls
* file1.txt file5.txt file7.txt
* ==> rm \*
* ==> touch pic{0..10..3}.{jpg,gif,jpeg}
* ==> ls
* pic0.gif pic0.jpg pic3.jpeg pic6.gif pic6.jpg pic9.jpeg
* pic0.jpeg pic3.gif pic3.jpg pic6.jpeg pic9.gif pic9.jpg
* ==> mkdir dir{1..5}
* ==> touch dir{1..5}/file{01..10}
* ==> ls dir3
* file01 file02 file03 file04 file05 file06 file07 file08 file09 file10
* ==> \_

### Tilde Expansion

* can be used as shortcut to some directory names
* ~ is home directory of current user
* ~username is home directory of specified user
* ~+ is the current directory, same as $PWD
* ~- is the previous directory, same as $OLDPWD
* will not be expanded within quotes
* here are some examples of tilde expansion:
* ==> echo ~
* /home/lczegel
* ==> echo "~"
* ~
* ==> index=~/public\_html/index.html
* ==> ls -l $index
* -rw-r--r-- 1 lczegel users 596 May 24 2013 /home/lczegel/public\_html/index.html
* ==> touch $index
* ==> ls -l $index
* -rw-r--r-- 1 lczegel users 596 Apr 7 22:50 /home/lczegel/public\_html/index.html
* ==> echo $PWD
* /home/lczegel/test/dir1
* ==> echo ~+
* /home/lczegel/test/dir1
* ==> echo $OLDPWD
* /home/lczegel
* ==> echo ~-
* /home/lczegel
* ==> file=~john.selmys/public/file16
* ==> echo $file
* /home/john.selmys/public/file16
* ==> \_
* can be used to find files within the directory stack
  + dirs - displays the directory stack
  + pushd dirname - pushes "dirname" onto directory stack and changes to it
  + popd - pops top directory name from directory stack and changes to next one
  + cd dirname - replaces top of directory stack with "dirname" and changes to it
  + the top of the directory stack is always the current directory
  + ~3 or ~+3 is replaced by the 3rd name from the top of the directory stack, starting at 0
  + ~-3 is replaced by the 3rd name from the bottom of the directory stack, starting at 0
* here are some examples of manipulating the directory stack:
* /home/lczegel: dirs
* ~
* /home/lczegel: cd test/dir1
* ~/test/dir1: dirs
* ~/test/dir1
* ~/test/dir1: pushd ~
* ~ ~/test/dir1
* /home/lczegel: pushd /etc
* /etc ~ ~/test/dir1
* /etc: pushd /var
* /var /etc ~ ~/test/dir1
* /var: cd
* /home/lczegel: dirs
* ~ /etc ~ ~/test/dir1
* /home/lczegel: echo ~3
* /home/lczegel/test/dir1
* /home/lczegel: echo ~-2
* /etc
* /home/lczegel: dirs
* ~ /etc ~ ~/test/dir1
* /home/lczegel: popd
* /etc ~ ~/test/dir1
* /etc: popd
* ~ ~/test/dir1
* /home/lczegel: popd
* ~/test/dir1
* ~/test/dir1: popd
* -bash: popd: directory stack empty
* ~/test/dir1: \_

### Command Substitution

* $(command-line) - new style (korn, bash) of command substitution, easily nested
* `command-line` - old style (bourne) of command substitution, nesting requires escaping inner back-quotes
* examples of nested command substitution:
* ==> echo -e "One year from today:\n\n $(cal $(date +'%m %Y' | awk '{print $1, $2 + 1}'))"
* One year from today:
* April 2016
* Su Mo Tu We Th Fr Sa
* 1 2
* 3 4 5 6 7 8 9
* 10 11 12 13 14 15 16
* 17 18 19 20 21 22 23
* 24 25 26 27 28 29 30
* ==> echo -e "One year from today:\n\n `cal \`date +'%m %Y' | awk '{print $1, $2 + 1}'\``"
* One year from today:
* April 2016
* Su Mo Tu We Th Fr Sa
* 1 2
* 3 4 5 6 7 8 9
* 10 11 12 13 14 15 16
* 17 18 19 20 21 22 23
* 24 25 26 27 28 29 30
* ==> \_

### Arithmetic Expansion

* $(( expression )) - new style (korn, bash) of arithmetic expansion, easily nested
* an example of arithmetic expansion:
* ==> echo -e "One year from today:\n\n $(cal $(date +'%m') $(($(date +'%Y') + 1)))"
* One year from today:
* April 2016
* Su Mo Tu We Th Fr Sa
* 1 2
* 3 4 5 6 7 8 9
* 10 11 12 13 14 15 16
* 17 18 19 20 21 22 23
* 24 25 26 27 28 29 30
* ==> \_

### Word Splitting

* shell splits results of previous expansions, that were not in double quotes, into words
* examples of word splitting:
* ==> var1=ford
* ==> grep $var1 cars
* ford mustang 65 45 17000
* ford ltd 83 15 10500
* ford thundbd 84 10 17000
* ford bronco 83 25 9525
* ==> var1="ford ltd"
* ==> grep $var1 cars
* grep: ltd: No such file or directory
* cars:ford mustang 65 45 17000
* cars:ford ltd 83 15 10500
* cars:ford thundbd 84 10 17000
* cars:ford bronco 83 25 9525
* ==> grep "$var1" cars
* ford ltd 83 15 10500
* ==> \_

### Shell Startup

### bash startup

* **/etc/profile** executes, then **.bash\_profile** in home directory
* **.bash\_login** is used if **.bash\_profile** doesn't exist
* **.profile** is used if **.bash\_profile** and **.bash\_login** don't exist
* **.bashrc** is used for interactive sub-shells

### ksh startup

* **/etc/profile** executes, then **.profile** in home directory
* **.kshrc** is used for interactive sub-shells

### alias

* assigns a new name to an existing utility
* eg. **alias** dir=ls
* eg. **alias** ls='ls -al'
* can be useful, but can also make your scripts cryptic