**UNX510/DPS918 - Unix BASH Shell Scripting**

**Lecture 3 - if Statement; Testing Conditions; Redirection**

**if Statement**

* "if" is used to conditionally execute one or more statements based on the exit status of the command following it
* basic syntax:
* if COMMAND
* then
* STATEMENT(S)

fi

- COMMAND will execute, then "if" will test the exit status of COMMAND, and the STATEMENT(S) will be executed if the exit status of COMMAND was zero

* an example:
* if cd $1
* then
* echo "Current directory has been changed to $PWD"

fi

**else**

* there is an optional "else" clause, for example:
* if cd $1
* then
* echo "Current directory has been changed to $PWD"
* else
* echo "Current directory is $PWD, could not be changed to $1"

fi

**elif**

* there is an optional "elif" clause, which slightly simplifies the syntax for nested if's
* the following two examples are logically identical:
* if grep "Lamborghini" $1
* then
* echo "We have at least one Lamborghini on the lot, hooray!"
* else if grep "Mercedes" $1
* then
* echo "We have at least one Mercedes on the lot, yippee!"
* else if grep "Toyota" $1
* then
* echo "We have at least one Toyota on the lot, whoopie!"
* else if grep "Fiat" $1
* then
* echo "We have at least one Fiat on the lot, too bad!"
* else
* echo "What happened to all our cars?"
* fi
* fi
* fi
* fi
* if grep "Lamborghini" $1
* then
* echo "We have at least one Lamborghini on the lot, hooray!"
* elif grep "Mercedes" $1
* then
* echo "We have at least one Mercedes on the lot, yippee!"
* elif grep "Toyota" $1
* then
* echo "We have at least one Toyota on the lot, whoopie!"
* elif grep "Fiat" $1
* then
* echo "We have at least one Fiat on the lot, too bad!"
* else
* echo "What happened to all our cars?"

fi

**Testing Conditions**

**test Command**

* "if" statements can be used in conjunction with the "test" command, which has the synonym "[ ... ]"
* "test" commands can also be used outside of "if" statements, as we'll see later when discussing "and lists" and "or lists"
* the following two examples are identical:
* if test $1 = "-l"
* then
* echo "A long listing will be produced:"
* fi
* ls $1
* if [ $1 = "-l" ]
* then
* echo "A long listing will be produced:"
* fi

ls $1

- note that the spaces are required, as shown, because "[ ... ]" is a command with several arguments   
- try running these without an argument, shows why you should usually double-quote arguments

* many programmers use "test" with "if" almost exclusively, even when it's not optimal
* the following two examples are logically identical, but the second version is much more efficient:
* grep "$1" cars
* if [ $? = 0 ]
* then
* echo "We have at least one $1 on the lot"
* fi
* if grep "$1" cars
* then
* echo "We have at least one $1 on the lot"

fi

* an example to check if the right number of arguments were passed to the script:
* if [ $# != 3 ]
* then
* echo "This script requires 3 arguments"
* exit 1

fi

**test Options**

* the test command has lots of options to test different conditions, as can be seen with a "man test" command
* to check if a filename exists, and you have read permission:
* if [ ! -r "$1" ]
* then
* echo "File $1 cannot be read"
* exit 1

fi

- note that ! means "not", inverting the sense of the test option

* or to check if a filename passed as an argument is an existing directory:
* if [ ! -d "$1" ]
* then
* echo "$1 is not a directory"
* exit 1

fi

* some very useful test options:
  + -f  - check if file is an ordinary file
  + -d  - check if file is a directory
  + -e  - check if file exists, either directory or ordinary file
  + -s  - check if file exists, with file size greater than 0
  + -r  - check if file exists, and you have read permission
  + -w  - check if file exists, and you have write permission
  + -x  - check if file exists, and you have execute permission
  + -nt - check if file on left is newer than file on right
  + -ot - check if file on left is older than file on right
  + -n  - check if string has non-zero length
  + -z  - check if string has zero length
  + -t  - check if specfied file descriptor is connected to the terminal
  + to compare numbers, can use -lt -le -gt -ge -eq -ne:   
      
    if [ $salary -gt 45000 ]   
    if test $salary -gt 45000   
      
    - note that \> and \< can be used for string comparisons, but they must be escaped, otherwise they would be confused with redirection
* conditions can be combined using the -a (and) and -o (or) options, for example:
* if [ ! -d "$1" -o ! -w "$1" -o ! -x "$1" ]
* then
* echo "Cannot delete a filename from $1"
* exit 1

fi

**Extended test Command**

* [[ ... ]] is a keyword rather than a command, so it is more efficient
* conditions can be combined using && (and) and || (or)
* will work correctly even if an unquoted variable is null
* string comparisons can use > and <, they won't be confused with redirection, for example:
* if [[ $1 > $2 || $2 > $3 ]]
* then
* echo "Arguments are not in correct sort order"
* exit 1

fi

* regular expression matches can be done using =~, for example:
* if [[ -z $1 || $1 =~ [^0-9] ]]
* then
* echo "First argument is not numeric"
* exit 1

fi

- pathname expansion is not performed within "[[ ... ]]" so the reg-exp doesn't need to be quoted

* here is an equivalent script without the extended test:
* if [ -z "$1" -o "$(echo $1 | grep "[^0-9]")" != "" ]
* then
* echo "First argument is not numeric"
* exit 1
* fi

- the reg-exp needs to be quoted, in case there is a single-character filename in the current directory

**Testing Using Shell Arithmetic**

* (( ... )) or let can be used
* if the expression evaluates to zero, then an exit status of 1 is returned, meaning false or failure
* if the expressions evaluates to non-zero, then an exit status of 0 is returned, meaning true or success
* operators &&, ||, >, >=, <, and <= can be used, for example:
* if (( $1 > $2 || $1 <= 0 ))
* then
* echo "Range of first two arguments is incorrect"
* exit 1

fi

**Redirection**

**Standard File Descriptors**

* standard input, output, and error are attached to your terminal, unless they are redirected
* standard input is also known as "stdin" and as "file descriptor 0"   
  - by default, command input is received from the terminal keyboard
* standard output is also known as "stdout" and as "file descriptor 1"   
  - by default, command output is sent to the terminal display
* standard error is also known as "stderr" and as "file descriptor 2"   
  - by default, command error messages are sent to the terminal display

**Standard Input**

* standard input (stdin) is file descriptor 0 (fd 0)
* used as the default input to any command
* defaults to the terminal keyboard, for example:
  + cat   
    - (by itself) will take input from terminal (use <Control>-d to end input), and send output and error messages to terminal
* < (or 0<) will redirect standard input from a file:
  + mail user < cars   
    - will redirect input from a file called "cars", instead of using keyboard input
  + tr '[a-z]' '[A-Z]' < cars   
    - will use "cars" as input
  + cat < cars   
    - will use "cars" as input
  + cat cars   
    - will use "cars" as input, because the "cat" executable will internally redirect the first argument to stdin - some commands do this, some don't
* << + will redirect following lines to standard input of a command, also called a "Here document", until + appears on a line by itself, + can be any combination of characters
  + an example:   
      
    cat << +   
    This technique is often used   
    to display a long error message   
    or a selection menu. Also, it   
    allows a small file to be internal   
    within a script, instead of using   
    a separate external file.   
    +

**Standard Output**

* standard output (stdout) is file descriptor 1 (fd 1)
* used as the default output from any command
* defaults to the terminal display
* > (or 1>) will redirect standard output to a file
  + if the file doesn't exist, it will be created:
    - cat cars > demofile   
      - will redirect output of the "cat" command to a new file called "demofile"
  + if the file does exist, any existing contents will be lost:
    - ls -l > demofile   
      - will redirect output of the "ls -l" command to "demofile", replacing existing contents
  + note that redirection is done by the shell BEFORE the command is called, so an existing file's contents are erased at that point:
    - grep ford demofile > demofile   
      - will remove contents of "demofile" before "grep" is called, and "demofile" will become empty
* >> (or 1>>) will also redirect standard output to a file
  + if the file doesn't exist, it will be created, same as with >
  + if the file does exist, output from the command will be appended (added) to the end of the file:
    - echo "This line will be appended" >> demofile   
      - will redirect output of the "cat" command to "demofile", appending to existing contents
* >/dev/null will redirect output to a null file
  + /dev/null is a special file which simply eats output, a black hole for command output
  + useful in scripting, when a command is used to do something, but you don't want the output displayed
  + find / -name \*.tmp > /dev/null   
    - will display only the error messages from the command
* /dev/null can also be used to empty a file
  + cat /dev/null > file1

**Standard Error**

* standard error (stderr) is file descriptor 2 (fd 2)
* used as the default error output from any command
* defaults to the terminal display
* 2> will redirect standard error to a file
  + if the file doesn't exist, it will be created:
    - cat missingfile 2> demofile   
      - will redirect error messages from the "cat" command to a new file
  + if the file does exist, any existing contents will be lost:
    - cd cars 2> demofile   
      - will redirect error message (about "cars" not being a directory) to "demofile", replacing existing contents
* 2>> will append standard error to a file, similar to the way >> works
* 2> /dev/null will redirect output to a null file, similar to the way 2>> works:
  + find / -name \*.tmp 2> /dev/null   
    - will display only command output, with no error messages

**Creating an Error Message**

* when writing a shell script, any error messages would normally be sent to stderr
* this way, your script's error messages will be treated the same as error messages from Linux commands
* >&2 (or 1>&2) will redirect standard output to standard error, this is how we create error messages in our scripts:
  + echo "This is a message"   
    echo "This is an error message" >&2   
    - if your script is executed using redirection, output and error messages will be redirected correctly   
    - note that & is used to redirect to a file descriptor, >2 would redirect to a file called "2"

**Redirecting Both stdout And stderr**

* stdout and stderr can be redirected to two different files:
  + cat cars > outputfile 2> errorfile
* there are several ways to redirect both standard output and standard error to the same file:
  + cat cars > demofile 2>&1   
    - redirects errors to file descriptor 1 (&1), which has already been redirected to "demofile"
  + cat cars 2> demofile >&2   
    - redirects output to file descriptor 2 (&2), which has already been redirected to "demofile"
  + cat cars &> demofile   
    - only works with the latest versions of the bash shell
  + note that the following will NOT work:   
    cat cars 2>&1 > demofile   
    cat cars >&2 2> demofile   
    - one file descriptor must be redirected to a file, before the second file descriptor is redirected to it

**Interactive Scripts**

* when creating an interactive shell script, sometimes output needs to be displayed at the terminal, regardless of any redirections
* this is especially important when asking for user input, otherwise the script will appear to the user to have stopped working, because the prompt message was redirected
* > /dev/tty (or 1> /dev/tty) will redirect standard output to the terminal:
  + echo -n "Please enter your phone number: " > /dev/tty   
    - this message will appear on the terminal display, regardless of any redirections of the containing script
* an example:
* echo -n "Please enter an integer: " > /dev/tty
* read number
* if [ -z "$number" ] || echo $number | grep "[^0-9]" > /dev/null
* then
* echo "Sorry, '$number' is not a valid integer" >&2
* else
* echo "Thank you!"
* fi