Chapter 6

Unix Shell Variables, Operators & Loop Control

School of Computer Science,
MIT-World Peace University,
Pune, Maharashtra, India.

Amol D. Vibhute (PhD)
Assistant Professor

Email ID:- amol.vibhute@mitwpu.edu.in

Roadmap of Chapter

- Variables,
- Special Variables,
- Command-Line Arguments,
- Special Parameters $* and $@,
- Exit Status,
- Using Shell Arrays,
- Basic Operators,
- Loop Control.

Introduction:

- A variable is a character string to which we assign a value.
- The value assigned could be a number, text, filename, device, or any other type of data.
- A variable is nothing more than a pointer to the actual data.
- The shell enables you to create, assign, and delete variables.
- **Variable Names**:
  - The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).
  - By convention, Unix shell variables will have their names in UPPERCASE.
  - The following examples are valid variable names −
    - _AMOL, TOKEN_A, VAR_1, VAR_2.
  - Following are the examples of invalid variable names −
    - 2_VAR, -VARIABLE, VAR1-VAR2, VAR_A!
- The reason you cannot use other characters such as !, *, or - is that these characters have a special meaning for the shell.

Cont.…

- **Defining Variables**:
  - Variables are defined as follows:
    - variable_name=variable_value
  - For example −
    - NAME="Amol Vibhute"
  - The above example defines the variable NAME and assigns the value "Amol Vibhute" to it. Variables of this type are called **scalar variables**. A scalar variable can hold only one value at a time.
  - Shell enables you to store any value you want in a variable. For example −
    - VAR1="Amol Vibhute"
    - VAR2=1007
- **Accessing Values**:
  - To access the value stored in a variable, prefix its name with the dollar sign ($) −
  - For example, the following script will access the value of defined variable NAME and print it on STDOUT −
    - #!/bin/sh
    - NAME="Amol Vibhute"
    - echo $NAME
  - Result:
    - $bash main.sh
    - Amol Vibhute

Cont.…

- Read-only Variables:
  - Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.
  - For example, the following script generates an error while trying to change the value of NAME −
    - #!/bin/sh
    - NAME="Amol Vibhute"
    - readonly NAME
    - NAME="Unix"
  - Result:
    - $bash main.sh
      main.bash: line 5: NAME: readonly variable
- Unsetting Variables:
  - Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable.
  - Following is the syntax to unset a defined variable using the unset command −
    - unset variable_name
  - The above command unsets the value of a defined variable. Here is a simple example that demonstrates how the command works −
    - #!/bin/sh
    - NAME="Amol Vibhutei"
    - unset NAME
    - echo $NAME
  - The above example does not print anything. You cannot use the unset command to unset variables that are marked **readonly**.

Special Variables:

- Previously, we understood how to be careful when we use certain nonalphanumeric characters in variable names. This is because those characters are used in the names of special Unix variables. These variables are reserved for specific functions.
- For example, the $ character represents the process ID number, or PID, of the current shell −
  - $echo $$
- The above command writes the PID of the current shell −
  - 1321
- The following table shows a number of special variables that you can use in your shell scripts −

Sr. No	Variable & Description
1	**$0**: The filename of the current script.
2	**$n**: These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).
3	**$#**: The number of arguments supplied to a script.
4	**$***: All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
5	**$@**: All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
6	**$?**: The exit status of the last command executed.
7	**$$**: The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
8	**$!**: The process number of the last background command.

Command-Line Arguments:

- The command-line arguments $1, $2, $3, ...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.

- Following script uses various special variables related to the command line-
  - #!/bin/sh
  - echo "File Name: $0"
  - echo "First Parameter : $1"
  - echo "Second Parameter : $2"
  - echo "Quoted Values: $@"
  - echo "Quoted Values: $*"
  - echo "Total Number of Parameters : $#"

- Here is a sample run for the above script −
  - File Name: si.sh
  - First Parameter : Amol
  - Second Parameter : Vibhute
  - Quoted Values: Amol Vibhute
  - Quoted Values: Amol Vibhute
  - Total Number of Parameters : 2

Special Parameters $* and $@:

- There are special parameters that allow accessing all the command-line arguments at once. **$*** and **$@** both will act the same unless they are enclosed in double quotes, "".
- Both the parameters specify the command-line arguments. However, the "$*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.

Exit Status:

- The **$?** variable represents the exit status of the previous command.
- Exit status is a numerical value returned by every command upon its completion. As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.
- Some commands return additional exit statuses for particular reasons. For example, some commands differentiate between kinds of errors and will return various exit values depending on the specific type of failure.
- Following is the example of successful command −
  - $./test.sh Amol Vibhute
  - File Name : ./test.sh
  - First Parameter : Amol
  - Second Parameter : Vibhute
  - Quoted Values: Amol Vibhute
  - Quoted Values: Amol Vibhute
  - Total Number of Parameters : 2
  - $echo $?
  - 0
  - $

Using Shell Arrays:

- A shell variable is capable enough to hold a single value. These variables are called **scalar variables**.
- Shell supports a different type of variable called an **array variable**. This can hold multiple values at the same time.
- Arrays provide a method of grouping a set of variables.
- Instead of creating a new name for each variable that is required, we can use a single array variable that stores all the other variables.
- All the naming rules discussed for Shell Variables would be applicable while naming arrays.
- Defining Array Values:
  - The difference between an array variable and a scalar variable can be explained as follows.
  - Suppose you are trying to represent the names of various students as a set of variables. Each of the individual variables is a scalar variable as follows −
    - NAME01="Amol"
    - NAME02="Santosh"
    - NAME03="Shourya"
    - NAME04="Advait"
    - NAME05="Arav"

Cont.…

- We can use a single array to store all the above-mentioned names. Following is the simplest method of creating an array variable. This helps assign a value to one of its indices.
- array_name[index]=value

- Here array_name is the name of the array, index is the index of the item in the array that you want to set, and value is the value you want to set for that item.
- As an example, the following commands −
    - NAME [0]="Amol"
    - NAME [1]="Santosh"
    - NAME [2]="Shourya"
    - NAME [3]="Advait"
    - NAME [4]="Arav"

- If you are using the ksh shell, here is the syntax of array initialization −
  - set -A array_name value1 value2 ... Value n
- If you are using the bash shell, here is the syntax of array initialization −
  - array_name=(value1 ... Value n)

Cont.…

- Accessing Array Values:
  - After you have set any array variable, you access it as follows −
    - ${array_name[index]}
- Here array_name is the name of the array, and index is the index of the value to be accessed. Following is an example to understand the concept −
  - #!/bin/sh
  - NAME=("Amol", "Santosh", "Shourya", "Advait", "Arav");
  - echo "First Index: ${NAME[0]}"
  - echo "Second Index: ${NAME[1]}"

    - Output: $ bash new.sh
      First Index: Amol
      Second Index: Santosh
  - We can access all the items in an array in one of the following ways −
    - ${array_name[*]}
    - ${array_name[@]}

Basic Operators:

- There are various operators supported by each shell. We will discuss in detail about Bourne shell (default shell).
- We will now discuss the following operators −
  - Arithmetic Operators
  - Relational Operators
  - Boolean Operators
  - String Operators
  - File Test Operators
- Bourne shell didn't originally have any mechanism to perform simple arithmetic operations, but it uses external programs, i.e., **expr**.

- The following example shows how to add two numbers −
  - #!/bin/sh
  - val=`expr 2 + 2`
  - echo "Total value : $val"
    - **Output**: Total value : 4
- The following points need to be considered while adding −
  - **There must be spaces between operators and expressions**. For example, 2+2 is not correct; it should be written as **2 + 2**.
  - The complete expression should be enclosed between ``, called the **backtick**.

Arithmetic Operators:

- The following arithmetic operators are supported by Bourne Shell.
- Assume variable **a holds 10** and variable **b holds 20** then −

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	`expr $a + $b` will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
* (Multiplication)	Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/ (Division)	Divides left hand operand by right hand operand	`expr $b / $a` will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
= (Assignment)	Assigns right operand in left operand	a = $b would assign value of b into a
== (Equality)	Compares two numbers, if both are same then returning true.	[$a == $b] would return false.
!= (Not Equality)	Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

Cont....

- It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example **[$a == $b]** is correct whereas, **[$a==$b]** is incorrect.
- All the arithmetical calculations are done using long integers.
- **Example**
  - Here is an example which uses all the arithmetic operators −

```
#!/bin/sh
a=10
b=20
val=`expr $a + $b`
echo "a + b : $val"
val=`expr $a - $b`
echo "a - b : $val"
val=`expr $a \* $b`
echo "a * b : $val"
val=`expr $b / $a`
echo "b / a : $val"
val=`expr $b % $a`
echo "b % a : $val"
if [ $a == $b ]
then
   echo "a is equal to b"
fi
if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

The above script will produce the following result −

```
$ bash Operators.sh
a + b : 30
a - b : -10
a * b : 200
b / a : 2
b % a : 0
a is not equal to b
```

The following points need to be considered when using the Arithmetic Operators −

- There must be spaces between the operators and the expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
- Complete expression should be enclosed between ‘ ‘, called the inverted commas.
- You should use \ on the * symbol for multiplication.

Relational Operators:

- Bourne Shell supports the following relational operators that are specific to numeric values. These operators do not work for string values unless their value is numeric.
- For example, following operators will work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".
- Assume variable **a holds 10** and variable **b holds 20** then −

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -le $b] is true.

Cont....

- **Example**
  - Here is an example which uses all the arithmetic operators −
  - #!/bin/sh

```
a=10
b=20
if [ $a -eq $b ]
then
   echo "$a -eq $b : a is equal to b"
else
   echo "$a -eq $b: a is not equal to b"
fi
if [ $a -ne $b ]
then
   echo "$a -ne $b: a is not equal to b"
else
   echo "$a -ne $b : a is equal to b"
fi
if [ $a -gt $b ]
then
   echo "$a -gt $b: a is greater than b"
else
   echo "$a -gt $b: a is not greater than b"
fi
if [ $a -lt $b ]
then
   echo "$a -lt $b: a is less than b"
else
   echo "$a -lt $b: a is not less than b"
fi
if [ $a -ge $b ]
then
   echo "$a -ge $b: a is greater or  equal to b"
else
   echo "$a -ge $b: a is not greater or equal to b"
fi
if [ $a -le $b ]
then
   echo "$a -le $b: a is less or  equal to b"
else
   echo "$a -le $b: a is not less or equal to b"
fi
```

The script will produce the following result −

- 10 -eq 20: a is not equal to b
- 10 -ne 20: a is not equal to b
- 10 -gt 20: a is not greater than b
- 10 -lt 20: a is less than b
- 10 -ge 20: a is not greater or equal to b
- 10 -le 20: a is less or equal to b

Boolean Operators:

- The following Boolean operators are supported by the Bourne Shell.
- Assume variable **a holds 10** and variable **b holds 20** then −

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical **OR**. If one of the operands is true, then the condition becomes true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false.	[$a -lt 20 -a $b -gt 100] is false.

String Operators:

- The following string operators are supported by Bourne Shell.
- Assume variable **a holds "abc"** and variable **b holds "efg"** then −

Operator	Description	Example
=	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n $a] is not false.
str	Checks if **str** is not the empty string; if it is empty, then it returns false.	[$a] is not false.

File Test Operators:

- We have a few operators that can be used to test various properties associated with a Unix file.
- Assume a variable **file** holds an existing file name "test" the size of which is 100 bytes and has **read**, **write** and **execute** permission on −

Operator	Description	Example
-b file	Checks if file is a block special file; if yes, then the condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file; if yes, then the condition becomes true.	[-c $file] is false.
-d file	Checks if file is a directory; if yes, then the condition becomes true.	[-d $file] is not true.
-f file	Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set; if yes, then the condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe; if yes, then the condition becomes true.	[-p $file] is false.

Cont.…

- t file	Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true.	[-t $file] is false.
-u file	Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable; if yes, then the condition becomes true.	[-r $file] is true.
-w file	Checks if file is writable; if yes, then the condition becomes true.	[-w $file] is true.
-x file	Checks if file is executable; if yes, then the condition becomes true.	[-x $file] is true.
-s file	Checks if file has size greater than 0; if yes, then condition becomes true.	[-s $file] is true.
-e file	Checks if file exists; is true even if file is a directory but exists.	[-e $file] is true.

Shell Decision Making:

- Unix Shell supports conditional statements which are used to perform different actions based on different conditions. Two decision-making statements are −
  - The **if...else** statement
  - The **case...esac** statement
- **The if...else statements**:
  - If else statements are useful decision-making statements which can be used to select an option from a given set of options.
  - Unix Shell supports following forms of **if…else** statement −
    - **if...fi statement**:
      - The **if...fi** statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally.
      - Syntax:
      - if [expression]
      - then
      - Statement(s) to be executed if expression is true
      - fi
    - If the resulting value is true, given statement(s) are executed. If the expression is false, then no statement would be executed. Most of the times, comparison operators are used for making decisions.

Cont.…

- **The if...else...fi statement**:
  - The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in a controlled way and make the right choice.
  - **Syntax**:
    - if [expression]
    - then
    - Statement(s) to be executed if expression is true
    - else
    - Statement(s) to be executed if expression is not true
    - fi
  - If the resulting value is true, given statement(s) are executed. If the expression is false, then no statement will be executed.

Cont.…

- **The if...elif...fi statement**:
  - The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.
  - Syntax:
    - if [expression 1]
    - then
    - Statement(s) to be executed if expression 1 is true
    - elif [expression 2]
    - then
    - Statement(s) to be executed if expression 2 is true
    - elif [expression 3]
    - then
    - Statement(s) to be executed if expression 3 is true
    - else
    - Statement(s) to be executed if no expression is true
    - fi
- This code is just a series of if statements, where each if is part of the else clause of the previous statement. Here statement(s) are executed based on the true condition, if none of the condition is true then else block is executed.

Cont.…

- **The case...esac Statement**:
  - You can use multiple **if...elif** statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.
  - Unix Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated **if...elif** statements.
  - Syntax: The basic syntax of the case...esac statement is to give an expression to evaluate and to execute several different statements based on the value of the expression.
  - The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.

```
case word in
  pattern1)
     Statement(s) to be executed if pattern1 matches
     ;;
  pattern2)
     Statement(s) to be executed if pattern2 matches
     ;;
  pattern3)
     Statement(s) to be executed if pattern3 matches
     ;;
  *)
    Default condition to be executed
     ;;
esac
```

  - Here the string word is compared against every pattern until a match is found. The statement(s) following the matching pattern executes. If no matches are found, the case statement exits without performing any action.
  - There is no maximum number of patterns, but the minimum is one.
  - When statement(s) part executes, the command ;; indicates that the program flow should jump to the end of the entire case statement.

Shell Loop Types:

- A loop is a powerful programming tool that enables you to execute a set of commands repeatedly.
- **The while Loop**:
  - The **while** loop enables you to execute a set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly.
  - Syntax:
    - while command
    - do
    - Statement(s) to be executed if command is true
    - done
  - Here the Shell command is evaluated. If the resulting value is true, given statement(s) are executed. If command is false, then no statement will be executed, and the program will jump to the next line after the done statement.

Cont.…

- **Nesting while Loops**:
- Example:

```
#!/bin/sh

a=0
while [ "$a" -lt 10 ]    # this is loop1
do
   b="$a"
   while [ "$b" -ge 0 ]  # this is loop2
   do
      echo -n "$b "
      b=`expr $b - 1`
   done
   echo
   a=`expr $a + 1`
done
```

This will produce the following result. It is important to note how **echo -n** works here. Here **-n** option lets echo avoid printing a new line character.

```
0
1 0
2 1 0
3 2 1 0
4 3 2 1 0
5 4 3 2 1 0
6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
8 7 6 5 4 3 2 1 0
9 8 7 6 5 4 3 2 1 0
```

Cont.…

- **The for Loop**:
  - The for loop operates on lists of items. It repeats a set of commands for every item in a list.
  - Syntax:
    - for var in word1 word2 ... wordN
    - do
    - Statement(s) to be executed for every word.
    - done
  - Here var is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

Shell Loop Control:

- The break statement
- The continue statement
- **The infinite Loop**:
  - All the loops have a limited life and they come out once the condition is false or true depending on the loop.
  - A loop may continue forever if the required condition is not met. A loop that executes forever without terminating executes for an infinite number of times. For this reason, such loops are called infinite loops.

```
#!/bin/sh
a=10
until [ $a -lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

This loop continues forever because **a** is always **greater than** or **equal to 10** and it is never less than 10.

Cont.…

- **The break Statement:**
  - The **break** statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement. It then steps down to the code following the end of the loop.
  - **Syntax:**
  - The following break statement is used to come out of a loop −
  - break
    - The break command can also be used to exit from a nested loop using this format −
  - break n
    - Here **n** specifies the **nth** enclosing loop to the exit from.

```
#!/bin/sh

a=0

while [ $a -lt 10 ]
do
   echo $a
   if [ $a -eq 5 ]
   then
      break
   fi
   a=`expr $a + 1`
done
```

Upon execution, you will receive the following result −

```
0
1
2
3
4
5
```

Cont.…

- **The continue statement:**
  - The **continue** statement is like the **break** command, except that it causes the current iteration of the loop to exit, rather than the entire loop.
  - This statement is useful when an error has occurred, but you want to try to execute the next iteration of the loop.
  - Syntax:
    - Continue
  - Like with the break statement, an integer argument can be given to the continue command to skip commands from nested loops.
    - continue n
    - Here **n** specifies the **nth** enclosing loop to continue from.

References:

- Unix Shell Programming: Yashwant Kanitkar, BPB Publications, New Delhi.

Thank You !!!