SYST13416
Introduction to Linux
Operating

Bash Shell Programming

SHELL OPERATORS

Objectives

- Bash shell operators
- Defining operators
- Evaluating operators
- Arithmetic operators
- Conditional operators

Programming process

The programming process consists of six steps (or stages):
1. ***Define***: involves planning and analysis, until the problem is well understood. (research, interviews, functional specifications)
2. ***Design***: involves problem solving and design of solution. (problem solving, brainstorming, Structured English, pseudocode, flowcharts, detailed specifications)
3. ***Develop***: involves selection of technologies and construction of the solution (coding)
4. ***Test***: a variety of testing, such as unit tests, logic tests, user testing, ...
5. ***Deploy***: involves putting the finished solution into production
6. ***Manage***: involves support and on-going maintenance of the product

More often than not, these steps may need to be repeated. For example, testing occurs after every step, or if a bug is found during debugging, it may be necessary to go back to the drawing board (design).

Important terms and concepts

object: a thing (nouns such as document, file, link, window, form, ...)
property: a property describes an object (adjectives such as colour, size, permissions, ...)
method: an instruction what you can do with an object (verbs such s open, close, copy,...)
statement: a statement combines objects, properties, and methods (just like a sentence in English)
function: a function is a collection of statements (just like a paragraph in English)
event: an event occurs when something happens (for example, a key is pressed)
variable: a variable is a location is computer memory that stores data temporarily. It is given a name to refer to it in the code.

Commands

Commands invoke programs that modify elements of the user experience, enable you to work with documents, executable code, and perform administrative tasks. There are several command families:

- Commands that find information
- Commands that channel input and output in non-standard ways
- Commands that help you work with the file system
- Commands that change and move existing files
- Commands that define and change file ownership and permissions
- Commands that set and enforce disk quotas

Operator

An operator is a type of function, just like in mathematics. An operator name or operator symbol is a notation which denotes a particular operator. Examples of operators are

- defining operators (assignment),
- evaluating operators (math, conditional statements), and
- redirecting and piping operators.

Scripting Languages

- Early scripting languages were created to shorten the traditional edit-compile-link-run process.
- Scripting language is a programming language that allows control of one or more software applications. Scripts are often interpreted from source code.
- Linux shells are scripting languages interpreted by the shell interpreters (sh, bash, csh, ksh, ...), whether used through an interactive session or written in script files (ASCII text file).
- The very beginning of a script file must indicate which interpreter (each has its own syntax) will understand the content of the file (the sha-bang line example for the Bourne shell interpreter is #!/bin/sh)

- A minimum content of a Bourne script template:

```
#!/bin/sh
#Program name: [name.sh]
#Author name: [Your Name]
#Date created: [Date the file was first created]
#Date updated: [Date the file was last modified]
#Description: [a short description of the script's purpose]
```

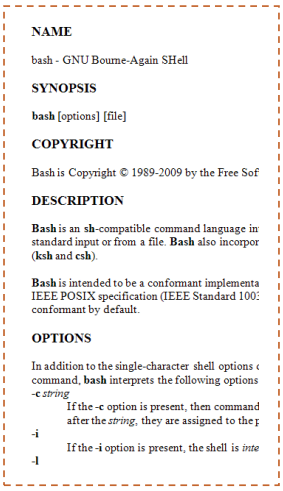
- Make sure you change the name of the interpreter for other shells, such as #!/bin/bash, #!/bin/csh, #!/bin/ksh, #!/usr/local/perl, and so on.
- Note that the system does not care whether or not the script file has an extension, BUT PEOPLE DO. It is part of the "Industry Best Practices" to use appropriate extensions, such as .sh for Bourne shell, .bsh for bash, .csh for C shell, .ksh for Korn, .pl for Perl, .php for PHP scripts, .js for Javascript, and so on.

- Your template can look like this:

```
#!/bin/bash
#Program name:
#Author name:
#Date created:
#Date updated:
#Description:
```

and saved as template.sh

- Another common "best practice" is to store all your scripts in a ~/bin subdirectory.



NAME

bash - GNU Bourne-Again SHell

SYNOPSIS

bash [options] [file]

COPYRIGHT

Bash is Copyright © 1989-2009 by the Free Sof

DESCRIPTION

Bash is an **sh**-compatible command language in
standard input or from a file. **Bash** also incorpor
(**ksh** and **csh**).

Bash is intended to be a conformant implementa
IEEE POSIX specification (IEEE Standard 100
conformant by default.

OPTIONS

In addition to the single-character shell options
command, **bash** interprets the following options
-c *string*
If the **-c** option is present, then command
after the *string*, they are assigned to the
-i
If the **-i** option is present, the shell is *inte*
-l

Bash Shell Operators

- The Bash shell operators are divided into three groups:
  - defining and evaluating operators
  - arithmetic operators
  - redirecting and piping operators

Defining Operators

- To create a variable and assign it a value, type at the prompt
  DOG=Poodle
- You created the variable named DOG and set its value to 'Poodle'
- CAUTION: There are no spaces between the = operator and its operands
- If value contains spaces and special characters it must be in quotes.

Evaluating Operators

- Type
  echo DOG
  → DOG
- Type
  echo $DOG
  → Poodle
- To set a variable to a string containing spaces, type
  MEMO="Meeting will be at noon today"

Redirection Operators (>, >>, 2>, <, <<)

- send output to something other than the standard output device
  - > redirection by creating a new file
  - >> appending to an existing file
  - 2> redirect stderr
- retrieve input from something other than the standard input device
  - < read information from a file instead

Preventing Redirection from Overwriting Files

- The > redirection operator overwrites an existing file
- Use the set command:
  set -o noclobber
- Once noclobber is set
  cat newFile > oldFile
  will not overwrite oldFile
- Once this option is set, to overwrite a file intentionally:
  cat newFile >! oldFile
  will overwrite oldFile

Example

- Force the ls command to display an error message by giving it an invalid argument. Assuming you have no file or directory in your home directory name jojo, type ls jojo and press Enter. You see the error message:
  ls: jojo: No such file or directory
- Redirect the error output of the ls command. Type
  ls jojo 2> errfile
  and press Enter. There is no output on the screen. Type cat errfile and press Enter. You see the output of the error in the file.

Example

- One method to get a sorted list of names of who is on the system is to type
  who > names.txt
  sort < names.txt
- A better way to do this is to connect the output of the who command directly to the input of the sort command. This will eliminate the need to remove the temporary file later.
  who | sort
  will give the same result as above, but quicker and cleaner.

The Pipe Operator (|)

- The pipe operator (|) redirects the output of one command to the input of another command. The pipe operator is used in the following manner:
  firstCommand | secondCommand
- The pipe operator connects the output of the first command with the input of the second command. The pipe operator can connect several commands on the same command line, in the following manner:
  firstCommand | secondCommand | thirdCommand ...

Preventing Redirection from Overwriting Files

- > redirection operator overwrites an existing file
- Use the set command:
  ```
  set -o noclobber
  ```
- Once noclobber is set
  ```
  cat newFile > oldFile
  ```
  will not overwrite oldFile
- Once this option is set, to overwrite a file intentionally:
  ```
  cat newFile >! oldFile
  ```
  will overwrite oldFile

Simple Arithmetic

Arithmetic Evaluation

- The shell allows arithmetic expressions to be evaluated, under certain circumstances (see the **let** and **declare** built-in commands and **Arithmetic Expansion**). Evaluation is done in fixed-width integers with no check for overflow, though division by 0 is trapped and flagged as an error. The operators and their precedence, associativity, and values are the same as in the C language.

Mathematical Operators

- The following list of operators is grouped into levels of equal-precedence operators. The levels are listed in order of decreasing precedence.

id++ id--	variable post-increment and post-decrement
++id –id	variable pre-increment and pre-decrement
- +	unary minus and plus
! ~	logical and bitwise negation
******	exponentiation
*** / %**	multiplication, division, remainder
+ -	addition, subtraction
<< >>	left and right bitwise shifts
<= >= < >	comparison
== !=	equality and inequality
&	bitwise AND
^	bitwise exclusive OR
|	bitwise OR
&&	logical AND
||	logical OR
expr?expr:expr	conditional operator
= *= /= %= += -= <<= >>= &= ^= |=	assignment
expr1 , expr2	comma

Declare Command

- Declare variables and/or give them attributes. If no *name*s are given then display the values of variables. The **-p** option will display the attributes and values of each *name*.
  - **-a** Each name is an indexed array variable (see **Arrays**).
  - **-A** Each name is an associative array variable (see **Arrays**).
  - **-i** The variable is treated as an integer; arithmetic evaluation is performed when the variable is assigned a value.

```
declare [-aAfFilrtux] [-p] [name[=value] ...]
```

Let Command

```
let arg [arg ...]
```

- Each arg is an arithmetic expression to evaluate. If the last arg evaluates to 0, let returns 1; 0 is returned otherwise.

Examples

- Display content of each variable. Are they the same? Why? Why not?

```
let x=14+5+4*8; echo $x
let y=(14+5)+4*8; echo $y
let z=(14+5+4)*8; echo $z
```

- Create the following script. What happens if you don't use the **let** command?

```
#!/bin/sh
#Program name: add.sh
#Description:
#  add 2 numbers and display each number and their sum
let A=1
let B=2
let sum=$A+$B
echo "A=$A, B=$B, sum=$sum"
```


Arithmetic Expansion

- Arithmetic expansion allows the evaluation of an arithmetic expression and the substitution of the result. The format for arithmetic expansion is:

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

- The expression is treated as if it were within double quotes, but a double quote inside the parentheses is not treated specially. All tokens in the expression undergo parameter expansion, string expansion, command substitution, and quote removal. Arithmetic expansions may be nested.

Conditional Statements

- Conditional statements are evaluated for true or false values.
- This is done with the test, or its equivalent, the [] operators.
- It the condition evaluates to true, a zero (TRUE) exit status is set,
- otherwise a non-zero (FALSE) exit status is set.
- If there are no arguments a non-zero exit status is set.

Tests for **strings**:

Test	Meaning
-z string	true if the string length is zero
-n string	true if the string length is non-zero
string1 = string2	true if string1 is identical to string2
string1 != string2	true if string1 is non identical to string2
string	true if string is not NULL

Integer comparisons

Test	Meaning
n1 -eq n2	true if integers n1 and n2 are equal
n1 -ne n2	true if integers n1 and n2 are not equal
n1 -gt n2	true if integer n1 is greater than integer n2
n1 -ge n2	true if integer n1 is greater than or equal to integer n2
n1 -lt n2	true if integer n1 is less than integer n2
n1 -le n2	true if integer n1 is less than or equal to integer n2

-

Test for files attributes

For filenames the options to test are given with the syntax:
[-option filename]

Test	Meaning
-r	true if it exists and is readable
-w	true if it exists and is writable
-x	true if it exists and is executable
-f	true if it exists and is a regular file
-d	true if it exists and is a directory
-h or	true if it exists and is a symbolic link
-c	true if it exists and is a character special file (i.e. the special device is accessed one character at a time)
-b	true if it exists and is a block special file (i.e. the device
-p	true if it exists and is a named pipe (fifo)
-u	true if it exists and is setuid (i.e. has the set-user-id bit
-g	true if it exists and is setgid (i.e. has the set-group-id
-k	true if it exists and the sticky bit is set (a t in bit 9)
-s	true if it exists and is greater than zero in size

Logical operators

Test	Meaning
!	negation (unary)
-a	and (binary)
-o	or (binary)
()	Expressions within the () are grouped together. You may need to quote the () to prevent the shell from

TEST AND EXPR COMMANDS

Objectives

- Command: **test**
- Command: **expr**

The **test** Command

- The test command makes preliminary checks of the Unix internal environment and other useful comparisons beyond those that the if command alone can perform. You can place the test command inside your shell script or execute it directly from the command line.
- The test command uses operators expressed as options to perform the evaluation. The command can be used to:
  - perform relational test with integers, such as equal, greater than, less than, etc.
  - test strings
  - determine if a file exists and what type of file it is
  - perform Boolean tests
- The test command returns a value known as an exit status. An exit status is a numeric value that the command returns to the operating system when it finishes. The Value of the test command's exit status indicates the results of the test performed. If the exit status is 0 (zero), the test result is true. An exit status of 1 indicates the test result is false.
- The exit status is normally detected in a script by the if statement or in a looping structure. You can view the last command's exit status by typing the command:
  echo $?

Examples

1. Create a variable number with the value 20 by typing the command
   number=20
   and pressing Enter. Type
   test $number -eq 20
   and press Enter. Type
   echo $?
   and press Enter. The result displayed on your screen should be 0, indicating true result.

2. Create a variable value with the value 10 by typing the command
   value=10
   and pressing Enter. Type
   test $number -lt $value
   and press Enter. Type
   echo $?
   and press Enter. The result displayed on your screen should be 1, indicating false result.

3. Try the test command's string testing capabilities:
   name="Linus"
   test $name = "Linus"
   echo $?
   test $name = "Barn"
   echo $?
   Observe the results. What do they mean?

4. Try the test command's file testing capabilities. Create an empty file name testFile:
   touch testFile
   Use the command
   ls -l testFile
   to view the file's permissions. Note that the file has read and write permissions for you, the owner. type
   test -x testFile
   echo $?
   The test command returns an exit status of 1 indicating that the testFile is not executable.
   test -r testFile
   echo $?
   The test command returns an exit status of 0 indicating testFile is readable.

The **expr** Command

- Enables you to perform simple math calculatins at the shell prompt. Itrecognizes all the arithmetic operators
- Those operators that have another meaning in the shell need to be prefixed by a backslash \ escape charater (for example, the * wildcard must be escaped for multiplication.)
- Each argument must be separate by whitespace.
- Parentheses can be used to establish operator precedence in longer expressions, must be escaped
- Can perform Boolean evalutions that return either true (1) or false (0)
- Can perform simple string manipulation with functions length, substr, and index

Examples

```
expr 7 + 3
10
expr 7 \* 3
21
expr 7 \* \( 3 + 1 \)
28
expr 7 = 3
0
expr length “Linux is the best”
17
expr substr “Linux is the best” 7 8
is the b
expr index “Linux is the best” “x”
5
```


Contents