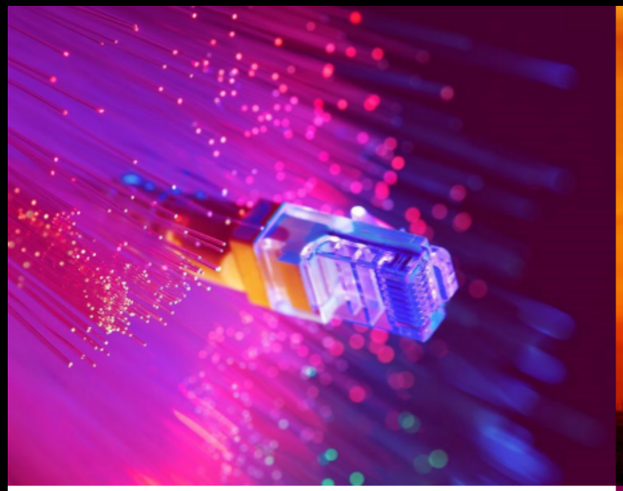

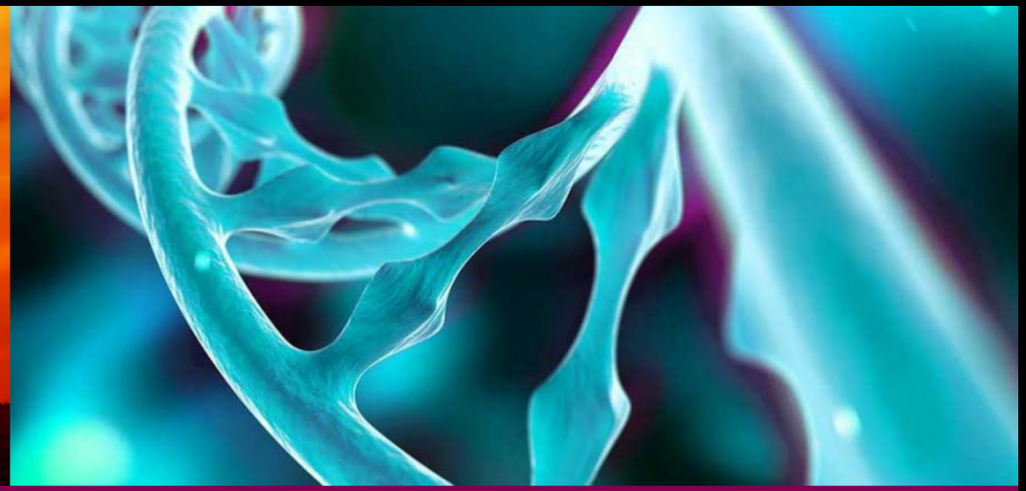

Working with bash — programming constructs

CSC Training, 2019-12


CSC – Finnish expertise in ICT for research, education and public administration

Comments

- A word beginning with # causes that word and all remaining characters on that line to be ignored.

```
$ echo "Have a nice day!" # no, it's not
$ #rm -rf Documents
```

- Exception: if commands are read from a file beginning with #!, the remainder of the first line specifies an interpreter for the program.

```
#!/bin/bash
echo "Have a nice day!"
# no, it's not
```

```
#!/usr/bin/python
print "Have a nice day!"
```

Parameters

- A *parameter* is an entity that stores values. A *variable* is a parameter denoted by a name.
  - In common speak, terms *parameter* and *variable* are interchangeable, but they are very different beings; parameters are set only by the shell, variables are user settable.

```
name=[value]
```

- A parameter (or variable) is set if it has been assigned a value. The null string is also a valid value.

Parameter expansion

- The `$` character introduces parameter expansion, command substitution, or arithmetic expansion.
- The basic form of parameter expansion is `${parameter}`. The value of *parameter* is substituted.
  - The braces are required when *parameter* is a positional parameter with more than one digit, or when *parameter* is followed by a character that is not to be interpreted as part of its name.

```
$ echo $HOME # --> echo /home/cscuser
$ echo ${HOME}s # --> echo /home/cscusers
```

Parameter expansion, substring expansion

```
$ echo ${string:-bummer} # --> echo bummer
$ echo ${string:=0123456789abcdefgh} # --> string=0123456789abcdefgh; echo 0123456789abcdefgh
$ echo ${string:7} # --> echo 7890abcdefgh
$ echo ${string: -7} # --> echo bcdefgh
$ echo ${string:7:2} # --> echo bc
$ echo ${!HO*} # --> echo HOME HOSTNAME HOSTTYPE
$ echo ${#HOME} # --> echo 13
$ echo ${HOME#/home} # --> echo /cscuser
$ echo ${HOME%user} # --> echo /home/csc
$ echo ${HOME^^} # --> echo /HOME/CSCUSER
$ echo ${HOME^^e} # --> echo /homE/cscusEr
```

```
$ echo $PATH; echo ${PATH/usr/abc}
$ echo $PATH; echo ${PATH//usr/abc}
$ echo $PATH; echo ${PATH//usr/}
$ echo $PATH; echo ${PATH//\/usr/}
```

Positional parameters

- *Positional parameters* are assigned from the command's arguments when it is invoked.
- A positional parameter is a parameter denoted by one or more digits, other than the single digit 0.
- Positional parameter *n* may be referenced as `${n}`, or as `$n` when *n* consists of a single digit.

```
$ mkdir -p one two/three # --> command="mkdir", $1="-p", $2="one", $3="two/three"
```

Special parameters

- `$@` expands to the positional parameters, starting from one.
- `$#` expands to the number of positional parameters.
- `$?` expands to the exit status of the most recently executed foreground pipeline.
- `$!` expands to the process ID of the job most recently placed into the background.
- `$$` expands to the process ID of the shell.
- `$0` Expands to the name of the shell or shell script.

Command substitution

- Command substitution allows the output of a command to replace the command itself.

```
$ list=$(ls exp)
$ echo $list
```

- Old-style backquote form is still in use, but it’s usage is strongly discouraged.

```
$ list=`ls exp`
$ echo $list
```

Arithmetic expansion

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

```
$ echo $(( 1+2*3**4 ))
```

- Supports only integers and (some of the) valid operators are:
  - exponentiation “**”;
  - multiplication “*”, division “/”, remainder “%”;
  - addition “+” and subtraction “-”;
  - comparisons “<= => < >”, equality “==” and inequality “!=”; and
  - bitwise AND “&”, XOR “^” and OR “|”, and logical AND “&&” and OR “||”.

Lists of commands

- A *list* is a sequence of one or more commands separated by one of the list operators.
  - Commands separated by a `;` are executed sequentially.
  - Commands separated by `&` are executed asynchronously.

```
$ ls; echo "Hello there!"
$ ls & echo "Hello there!"
```

- AND and OR lists are sequences of one or more comands separated by the control operators `&&` and `||`, respectively.
  - `command1 && command2`: *command2* is executed if *command1* succeeds.
  - `command1 || command2`: *command2* is executed if *command1* fails.

```
$ mkdir foo && echo "Success"
$ mkdir foo || echo "Fail"
```

Compound commands

- Compound commands are the shell programming language constructs.
- Each construct begins with a reserved word or control operator and is terminated by a corresponding reserved word or operator.
- Bash provides conditional constructs, looping constructs, and mechanisms to group commands and execute them as a unit.

Conditional constructs

- The syntax of the if command is:

```
if test-commands; then
  consequent-commands;
[elif more-test-commands; then
  more-consequents;]
[else
  alternate-consequents;]
fi
```

Conditional expressions

- The *test-commands* often involves numerical or string comparison tests, but it can be any command that returns a status of zero when it succeeds, and some other status when it fails.
- Bash provides two comparison expressions:
  - command `test` (and it's synonym `[`), which is more portable but limited in features; and
  - the "new test" command `[[`, which has fewer surprises and is generally safer to use — but works only in Bash.

```
$ if [ "$LOGNAME" = "cscuser" ]; then
>     echo "Love you!";
> fi
```

Conditional expressions

```
$ if [ "$LOGNAME" = "cscuser" ]; then echo "Love you!"; fi
$ if test "$LOGNAME" = "cscuser"; then echo "Love you!"; fi
$ test "$LOGNAME" = "cscuser" && echo "Love you!"
$ [ "$LOGNAME" = "cscuser" ] && echo "Love you!"

$ if [ "$LOGNAME" = "cscuser" ]; then echo "Love you!"; else echo "How do you do!"; fi
$ [ "$LOGNAME" = "cscuser" ] && echo "Love you!" || echo "How do you do!"

$ [ -e "Desktop" ] && echo "File Desktop exists."
$ [ "Desktop" -nt "exp" ] && echo "Desktop is newer" || echo "exp is newer"

$ [ $(id -u) -ne $UID ] && echo "User ID's do not match."
$ [ $(( $(id -u) + 1 )) -ne $UID ] && echo "User ID's do not match."

$ if [ -e Desktop -a -d Desktop ]; then echo "Desktop is a directory"; fi
$ [ -e Desktop && -d Desktop ] && echo "Desktop is a directory" # This does not work
$ [[ -e Desktop && -d Desktop ]] && echo "Desktop is a directory" # But this does work
```

Looping constructs

- Bash supports the following looping constructs (note that a `;` may be replaced with a newline):
  - `until` *test-commands* `;` `do` *consequent-commands* `;` `done`
    Execute *consequent-commands* as long as *test-commands* has an exit status which is not zero.
  - `while` *test-commands*`;` `do` *consequent-commands*`;` `done`
    Execute *consequent-commands* as long as *test-commands* has an exit status of zero.
  - `for` *name* `[ [in [`*words* …`] ] ; ]` `do` *commands*`;` `done`
    Expand *words* and execute *commands* once for each member in the resultant list, with *name* bound to the current member.

Looping constructs

```
$ for sqr in 1 2 3 4 5 6; do echo $(( 2 ** $sqr )); done
$ for sqr in {1..6}; do echo $(( 2 ** $sqr)); done
$ for (( sqr = 1; sqr <= 6; sqr++ )); do echo $(( 2 ** $sqr )); done
```

```
$ for sqr in {1..6..2}; do echo $(( 2 ** $sqr )); done
```

```
$ for i in *; do echo "$i"; done # Equivalent to 'ls -1'
```

```
$ for ns in $(grep ^nameserver /etc/resolv.conf); do
>   if [ $ns = nameserver ]; then continue; fi
>   ping -c1 $ns
> done
$ for ns in "$(grep ^nameserver /etc/resolv.conf)"; do ping -c1 ${ns##nameserver }; done
```

Looping constructs

```
while read -r line; do
  echo "::: $line"
  [[ "$line" == "end" ]] && break
done
```

```
until [ "$line" == "end" ]; do
  read -r line
  echo "::: $line"
done
```

Command grouping

- Bash provides two ways to group a list of commands to be executed as a unit.

  `( *list* )`

  Placing a *list* of commands between parentheses causes a subshell environment to be created, and each of the commands in *list* to be executed in that subshell.

  `{ *list*; }`

  Placing a *list* of commands between curly braces causes the *list* to be executed in the current shell context. The semicolon (or newline) following *list* is required.

```
$ ( echo "Directory listing:"; ls ) > dir_list
```

Functions

- Shell functions are a way to group commands for later execution using a single name for the group.
- Functions are declared using either of these these syntax:
  name `()` *compound-command* `[` *redirections* `]`
  `function` *name* `[()]` *compound-command* `[` *redirections* `]`
- When a function is executed, the arguments to the function become the positional parameters during its execution.

```
$ ll () { ls -laF "$@" | more; }
```

Command execution

- When a simple command is executed, the shell performs the following expansions, assignments, and redirections, from left to right.
  1. The words that the parser has marked as variable assignments and redirections are saved for later processing.
  2. The words that are not variable assignments or redirections are expanded. If any words remain after expansion, the first word is taken to be the name of the command and the remaining words are the arguments.
  3. Redirections are performed.
  4. The text after the ‘=’ in each variable assignment undergoes expansions.
- If no command name results, the variable assignments affect the current shell environment. Otherwise, the variables are added to the environment of the executed command.

Exit status

- On UNIX systems, every command must return an exit status value, which is returned to the shell and they fall between 0 and 255.
- For the shell’s purposes, a command which exits with a zero exit status has succeeded. A non-zero exit status indicates failure.
  - If a command is not found, the child process created to execute it returns a status of 127. If a command is found but is not executable, the return status is 126.
  - Generally, an exit status of 2 indicates incorrect usage, generally invalid options or missing arguments.