

Administration of Operating Systems

DO2003

<http://www.hh.se/do2003>

Programming the Bourne Again Shell



Bourne Again Shell (bash)

- Executes commands from the command line and from shell scripts files
- A shell script is a text file containing
 - Shell commands, control flow structures, parameters, variables, functions, ...
- Running scripts
 1. `$ Filename [arguments]`
 2. `$./Filename [arguments]`
 3. `$ bash Filename [arguments]`

Very simple script

```
$ PATH=$PATH:/home/ide
```

← working directory in the PATH

```
$ chmod 700 *.sh
```

← rwx permission for all scripts

```
$ cat > srm
```

```
#!/bin/bash
```

```
# Safe Remove
```

```
backup_path=/tmp
```

```
cp $1 $backup_path
```

```
echo $1 is stored in $backup_path
```

```
rm $1
```

```
$ chmod u+x srm
```

```
$ srm report.txt
```

```
report.txt is stored in /tmp
```

```
$
```

Parameters and variables

- A parameter stores values and a variable is a parameter denoted by a name

- Users can define *User-created variables*

VARIABLE=value

- The shell has *keyword variables*

- Array variables

VARIABLE=(element1 element2 ...)

```
$ STUDENTS=(Wagner Slawomir Mattias)
```

```
$ echo ${STUDENTS[1]}
```

```
Slawomir
```

```
$ echo ${#STUDENTS[*]}
```

```
3
```

```
$ echo ${#STUDENTS[1]}
```

```
8
```

Special parameters

- \$? Exit status or exit code
 - 0 is success (true)
- \$# Number of Command-Line Arguments
- \$0 Name of the Calling Program
- \$1–\$n Command-Line Arguments
- shift Promotes Command-Line Arguments
- set Initializes Command-Line Arguments
- \$* and \$@ Represent All Command-Line Arguments

Conditional construct: `if...then...else`

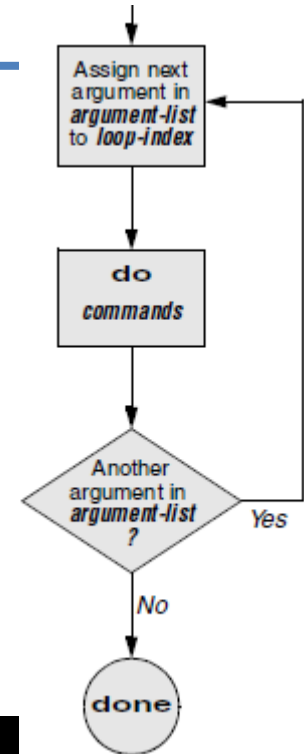
```
## If statement control structure
if [ <condition> ]
then
    #This code is executed only if the condition evaluates to true
fi
## If statement control structure
if [ <condition> ]
then
    #This code is executed only if the condition evaluates to true
else
    #This code is executed only if the condition evaluates to false
fi
#Arithmetic relational operators
#-eq
#-ne
#-gt
#-ge
#-lt
#-le
#example value -eq 10
```

Looping Constructs: **for**

- Perform a loop over the argument list

```
$ cat count.sh
#!/bin/bash
for (( i=1; i<=5; i++ ))
do
    echo "Counting $i times..."
done
$
```

```
$ cat count2.sh
#!/bin/bash
for i in 1 2 3 4 5
do
    echo "Counting $i times..."
done
$
```



Break and continue control structures

- Both alter control within loops
 - break transfers control out of a loop
 - exit the current loop before its normal ending
 - continue transfers control immediately to the top of a loop

```
$ cat brkcont.sh
#!/bin/bash
for index in 1 2 3 4 5 6 7 8 9 10
do
    if [ $index -le 3 ] ; then
        echo "continue"
        continue
    fi
    echo $index
    if [ $index -ge 8 ] ; then
        echo "break"
        break
    fi
done
$
```

File descriptors

- Place that a program can send its output to and get its input from
- Known file descriptors
 - stdin (0), stdout (1) and stderr (2)
- When a process opens a file, Linux associates a number, the file descriptor, with the file
 - exec n> outfile
 - exec m< infile
- Read and write operations use the file descriptor
- At the, close the file descriptor

Operator	Meaning
<&m	Duplicates stdin from file descriptor m
[n]>&m	Duplicates stdout or file descriptor n if specified from file descriptor m
[n]<&-	Closes standard input or file descriptor n if specified
[n]>&-	Closes standard output or file descriptor n if specified

File descriptors

```
$ cat FileDesc.sh
#!/bin/bash
usage ()
{
if [ $# -ne 2 ]; then
    echo "Usage: $0 file1 file2" 2>&1
    exit 1
fi
}
usage "$@"
exec 3<$1
exec 4<$2
read line1 <&3
echo "$line1"
read line2 <&4
echo "$line2"
read -u3 lineword1 lineword2 lineremaining
echo "$lineword1, $lineword2, $lineremaining"
exec 3<$- 4<$-
$
```

File descriptors

```
#!/bin/bash
# Redirecting stdin using 'exec'.
exec 6<&0      # Link file descriptor #6 with stdin. Saves stdin
exec < $1      # stdin replaced by file argument 1
read a1       # Reads first line of file argument 1
read a2       # Reads second line of file argument 1
echo "Following lines read from file."
echo $a1echo $a2echo;
exec 0<&6 6<&- # Restores stdin from fd #6 and close fd #6 ( 6<&- )
echo -n "Enter data "
read userInput # "read" reads from normal stdin.
echo "Input read from stdin. "
echo "User Input = $userInput"
echo
```

http://linuxtopia.org/online_books/advanced_bash_scripting_guide/x13082.html

Debugging shell scripts

- Programming mistakes are common
- The `-x` option helps to debug a script, i.e, traces a script's execution
- The shell displays each command before it runs the command

```
$ bash -x compIN1.sh wagner wagner
+ test wagner = wagner
+ echo 'Inputs are equal'
+ exit 0
$
```

Builtins

- `type` Displays how each argument would be interpreted as a command
- `read` Reads a line from standard input
- `exec` Executes a command or redirect file descriptors
 - If used to execute a command within a script, `exec` does not return the control to the script
- `trap` Catches signals. A script can take actions when it receives a specified signal
- `kill` Sends a signal to a process or job
- `getopts` Parses arguments to a shell script
- `let` Allows arithmetic to be performed on shell variables
 - The use of `$` is not need in front of a variable

Let's create a simple script

```
#!/bin/bash
# We are going to check if a customer is allowed to buy at systembolaget
# Add a variable to store the client's age
# Ask the client's age and store it into the variable
# Check if the client is older than 20
# If the client is older than 20 display 'You are allowed to buy' otherwise display
# 'We do not sell alcohol to anyone under the age of 20!'
# If the client is older than 20 and younger than 25, display 'Show your
# identification before a purchase'
```

Help with BASH programming

- <http://tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf>
- http://linuxconfig.org/Bash_scripting_Tutorial
- <http://www.gnu.org/software/bash/manual/>
- <http://www.digilife.be/quickreferences/QRC/Bash%20Quick%20Reference.pdf>

