

# Shell Programming—an Introduction

Copyright Conditions: Open Publication License (see  
<http://www.opencontent.org/openpub/>)

Nick Urbanik  
nicku@nicku.org  
A computing department

Intro	
Aim . . . . .	slide 2
Why Shell Scripting? . . . . .	slide 3
Where to get more information . . . . .	slide 4
The Shell is an Interpreter . . . . .	slide 5
The Shebang . . . . .	slide 6
Making Executable . . . . .	slide 7
Quoting and Funny Chars	
Special Characters . . . . .	slide 8
Special Chars—2 . . . . .	slide 9
Special Chars—3 . . . . .	slide 10
Quoting . . . . .	slide 11
Quoting—2 . . . . .	slide 12
Quoting—When to use it? . . . . .	slide 13
Variables	
True and False . . . . .	slide 14
Variables—1 . . . . .	slide 15
Variables—Assignments . . . . .	slide 16
Variables—Local to Script . . . . .	slide 17
Variables—Unsetting Them . . . . .	slide 18
Command-line Parameters . . . . .	slide 19
Special Built-in Variables . . . . .	slide 20
Variables: use Braces $\$\{ \dots \}$ . . . . .	slide 21
After $\$9$ . . . . .	slide 22
More about Quoting . . . . .	slide 23
Command Substitution	
Command Substitution . . . . .	slide 24
Example of Cmd Subst . . . . .	slide 25
Conditions	
Comparing Strings . . . . .	slide 26
Comparing Integers . . . . .	slide 27
File Tests & NOT . . . . .	slide 28
AND OR Conditions . . . . .	slide 29
Arithmetic	
Arithmetic Assignments . . . . .	slide 30
$\$( \dots )$ . . . . .	slide 31
$( \dots )$ . . . . .	slide 32

## Statements

<code>if</code> Statement. . . . .	slide 33
<code>while</code> Statement. . . . .	slide 34
<code>for</code> Statement. . . . .	slide 35
<code>for</code> Loops: Another Example. . . . .	slide 36
<code>for</code> (( ; ; )) . . . . .	slide 37
<code>break</code> and <code>continue</code> . . . . .	slide 38
Blocks: { . . . } . . . . .	slide 39
Flow Control: <code>  </code> && . . . . .	slide 40

## Input & Output

Output: <code>echo</code> and <code>printf</code> . . . . .	slide 41
Input: the <code>read</code> Command. . . . .	slide 42
Split with <code>set</code> . . . . .	slide 43
More about <code>set</code> , and <code>IFS</code> . . . . .	slide 44
Example: Changing <code>IFS</code> . . . . .	slide 45

## Command-line Parameters

<code>case</code> Statement. . . . .	slide 46
<code>case</code> Statement: Example . . . . .	slide 47
<code>shift</code> Up . . . . .	slide 48
<code>shift</code> : Many Places. . . . .	slide 49
Command-Line Options—1 . . . . .	slide 50
Command-Line Options—2 . . . . .	slide 51
Command-Line Options—3 . . . . .	slide 52
<code>getopts</code> —4. . . . .	slide 53
<code>getopts</code> —5. . . . .	slide 54

## Temporary Files, Signals

Temporary Files: <code>mktemp</code> . . . . .	slide 55
Signals that Kill . . . . .	slide 56
Signals: <code>trap</code> . . . . .	slide 57
Signals: <code>trap</code> Example. . . . .	slide 58

## Functions

Functions . . . . .	slide 59
Parameters in Functions. . . . .	slide 60
Example, Calling a Function. . . . .	slide 61

## Debugging

Debugging Shell Scripts—1. . . . .	slide 62
Debugging Shell Scripts—2. . . . .	slide 63

Writing Shell Scripts . . . . .	slide 64
Useful External Programs—1 . . . . .	slide 65
Useful External Programs—2 . . . . .	slide 66
Regular Expressions	
Regular Expressions. . . . .	slide 67
What is In a RegEx? . . . . .	slide 68
Literal characters. . . . .	slide 69
Character Classes: [...] . . . . .	slide 70
[^...] . . . . .	slide 71
Match Any Character . . . . .	slide 72
Match Start or End . . . . .	slide 73
Repetitions . . . . .	slide 74
Matching Alternatives: “ ” . . . . .	slide 75
Examples . . . . .	slide 76
<b>awk</b> and <b>sed</b>	
Basic <b>awk</b> . . . . .	slide 77
What Does <b>awk</b> Do? . . . . .	slide 78
<b>awk</b> Examples . . . . .	slide 79
<b>sed</b> —the Stream Editor . . . . .	slide 80
<b>sed</b> —Backreferencees. . . . .	slide 81
Backrefs Example . . . . .	slide 82
<b>find</b>	
<b>find</b> Examples . . . . .	slide 83
Finding SUID Programs. . . . .	slide 84
Long <b>find</b> Example. . . . .	slide 85
<b>rpm</b> Queries. . . . .	slide 86

## Aim

After successfully working through this exercise, You will:

- write simple shell scripts using `for`, `if`, `while`, `case`, `getopts` statements;
- write shell script functions, and be able to handle parameters;
- understand basic regular expressions, and be able to create your own regular expressions;
- understand how to execute and debug these scripts;
- understand some simple shell scripts written by others.

## Why Shell Scripting?

- Basic startup, shutdown of Linux, Unix systems uses large number of shell scripts
  - understanding shell scripting important to understand and perhaps modify behaviour of system
- Very high level: powerful script can be very short
- Can build, test script incrementally
- Useful on the command line: “one liners”

## Where to get more information

- the Library has two copies of the book, *Learning the Bash Shell*, second edition, by Cameron Newham & Bill Rosenblatt, O'Reilly, 1998.
- There is a free on-line book about shell programming at: <http://tldp.org/LDP/abs/html/index.html> and <http://tldp.org/LDP/abs/abs-guide.pdf>. It has hundreds of pages, and is packed with *examples*.
- The handy reference to shell programming is:

```
$ pinfo bash
```

or

```
$ man bash
```

- **IMPORTANT**: `bash` provides simple on-line help for all built-in commands, e.g.,

```
$ help let
```

## The Shell is an Interpreter

- Some languages are compiled: C, C++, Java,...
- Some languages are interpreted: Java bytecode, Shell
- Shell is an interpreter: kernel does not run shell program directly:
  - kernel runs the shell program `/bin/sh` with script file name as a parameter
  - the kernel cannot execute the shell script directly, as it can a binary executable file that results from compiling a C program

## The Shebang

- You ask the Linux kernel to execute the shell script
- kernel reads first two characters of the executable file
  - If first 2 chars are “#!” then
  - kernel executes the name that follows, with the file name of the script as a parameter
- Example: a file called `find.sh` has this as the first line:

```
#!/bin/sh
```

- then kernel executes this:

```
/bin/sh find.sh
```

- What will happen in each case if an executable file begins with:
  - `#! /bin/rm`
  - `#! /bin/ls`

## Making the script executable

To easily execute a script, it should:

- be on the PATH
- have execute permission.

How to do each of these?

- Red Hat Linux by default, includes the directory `~/bin` on the PATH, so create this directory, and put your scripts there:

```
$ mkdir ~/bin
```

- If your script is called `script`, then this command will make it executable:

```
$ chmod +x script
```

## Special Characters

- Many characters are *special* to the shell, and have a particular meaning to the shell.

Character	Meaning	See slide
~	Home directory	§ 7
`	Command substitution. Better: <code>\$(...)</code>	§ 24
#	Comment	
\$	Variable expression	§ 15
&	Background Job	2.10 on page 41
*	File name matching wildcard	2.18 on page 49
	Pipe	2.9 on page 40



## Special Characters—continued: 2

Character	Meaning	See slide
(	Start subshell	§ 45, 17, 39
)	End subshell	§ 45, 17, 39
[	Start character set file name matching	2.9 on page 40
]	End character set file name matching	2.9 on page 40
{	Start command block	§ 39
;	Command separator	§ 40
\	Quote next character	§ 23
'	Strong quote	§ 23
"	Weak quote	§ 23

## Special Characters—continued: 3

Character	Meaning	See slide
<	Input redirect	2.7 on page 38
>	Output redirect	2.6 on page 37
/	Pathname directory separator	
?	Single-character match in filenames	2.18 on page 49
!	Pipeline logical NOT	§ 28
<i>&lt;space or tab&gt;</i>	shell normally splits at white space	§ 44

- Note that references to pages in the tables above refer to the modules in the workshop notes

## Quoting

- Sometimes you want to use a special character *literally*; i.e., without its special meaning.
- Called *quoting*
- Suppose you want to print the string: `2 * 3 > 5 is a valid inequality?`
- If you did this:

```
$ echo 2 * 3 > 5 is a valid inequality
```

the new file `'5'` is created, containing the character `'2'`, then the names of all the files in the current directory, then the string `"3 is a valid inequality"`.

## Quoting—2

- To make it work, you need to protect the special characters `'*` and `'>` from the shell by quoting them. There are three methods of quoting:
  - Using double quotes (“weak quotes”)
  - Using single quotes (“strong quotes”)
  - Using a backslash in front of each special character you want to quote
- This example shows all three:

```
$ echo "2 * 3 > 5 is a valid inequality"
$ echo '2 * 3 > 5 is a valid inequality'
$ echo 2 \* 3 \> 5 is a valid inequality
```

## Quoting—When to use it?

- Use quoting when you want to pass special characters to another program.
- Examples of programs that often use special characters:
  - `find`, `locate`, `grep`, `expr`, `sed` and `echo`
- Here are examples where quoting is required for the program to work properly:

```
$ find . -name \*.jpg
$ locate '/usr/bin/c*'
$ grep 'main.*( ' *.c
$ i=$(expr i \* 5)
```

## True and False

- Shell programs depend on executing external programs
- When any external program execution is successful, the exit status is zero, 0
- An error results in a non-zero error code
- To match this, in shell programming:
  - The value 0 is true
  - any non-zero value is false
- This is opposite from other programming languages

## Variables—1

- Variables not declared; they just appear when assigned to
- Assignment:
  - no dollar sign
  - no space around equals sign
  - examples:
    - `$ x=10 # correct`
    - `$ x = 10 # wrong: try to execute program called “x”`
- Read value of variable:
  - put a ‘\$’ in front of variable name
  - example:
    - `$ echo "The value of x is $x"`

## Variables—Assignments

- You can put multiple assignments on one line:
  - `i=0 j=10 k=100`
- You can set a variable temporarily while executing a program:
  - `$ echo $EDITOR`  
`emacsclient`
  - `$ EDITOR=gedit crontab -e`
  - `$ echo $EDITOR`  
`emacsclient`

## Variables—Local to Script

- Variables disappear after a script finishes
- Variables created in a sub shell disappear
  - parent shell cannot read variables in a sub shell
  - example:

```
$ cat variables
#!/bin/sh
echo $HOME
HOME=happy
echo $HOME
$ ./variables
/home/nicku
happy
$ echo $HOME
/home/nicku
```

## Variables—Unsetting Them

- You can make a variable hold the null string by assigning it to nothing, but it does not disappear totally:

```
$ VAR=
$ set | grep '^VAR'
VAR=
```

- You can make it disappear totally using **unset**:

```
$ unset VAR
$ set | grep '^VAR'
```

## Command-line Parameters

- Command-line parameters are called \$0, \$1, \$2, ...
- Example: when call a shell script called “shell-script” like this:

```
$ shell-script param1 param2 param3 param4
```

---

<i>variable</i>	<i>value</i>
\$0	shell-script
\$1	param1
\$2	param2
\$3	param3
\$4	param4
\$#	number of parameters to the program, e.g., 4

---

- Note: these variables are read-only.

## Special Built-in Variables

- Both `$@` and `$*` are a list of all the parameters.
- The only difference between them is when they are quoted in quotes—see manual page for `bash`
- `$?` is exit status of last command
- `$$` is the process ID of the current shell
- Example shell script:

```
#!/bin/sh
echo $0 is the full name of this shell script
echo first parameter is $1
echo first parameter is $2
echo first parameter is $3
echo total number of parameters is $#
echo process ID is $$
```

## Variables: use Braces `${...}`

- It's good to put braces round a variable name when getting its value
- Then no problem to join its value with other text:

```
$ test=123
$ echo ${test}
123
# No good, variable $test456 is undefined:
$ echo $test456

$ echo ${test}456
123456
```

## Braces and Parameters after \$9

- Need braces to access parameters after \$9:

```
$ cat paramten
#!/bin/sh
echo $10
echo ${10}
$ ./paramten a b c d e f g h i j
a0
j
```

- Notice that \$10 is the same as \${1}0, i.e., the first parameter “a” then the literal character zero “0”

## More about Quoting

- **Double quotes**: “...” stop the special behaviour of all special characters, except for:
  - variable interpretation (\$)
  - backticks (`) — see slide 24
  - the backslash (\)
- **Single quotes** ‘...’:
  - stop the special behaviour of *all* special characters
- **Backslash**:
  - preserves literal behaviour of character, except for newline; see slides §29, §31, §35
  - Putting “\” at the end of the line lets you continue a long line on more than one physical line, but the shell will treat it as if it were all on one line.



## Command Substitution — `$(...)` or `'...'`

- Enclose command in `$(...)` or backticks: `'...'`
- Means, “Execute the command in the `$(...)` and put the output back here.”
- Here is an example using `expr`:

```
$ expr 3 + 2
5
$ i=expr 3 + 2    # error: try execute command '3'
$ i=$(expr 3 + 2) # correct
$ i='expr 3 + 2'  # also correct
```

## Command Substitution—Example

- We want to put the *output of the command* `hostname` into a *variable*:

```
$ hostname
nickpc.tyict.vtc.edu.hk
$ h=hostname
$ echo $h
hostname
```

- Oh dear, we only stored the *name* of the command, not the *output* of the command!
- *Command substitution* solves the problem:

```
$ h=$(hostname)
$ echo $h
nickpc.tyict.vtc.edu.hk
```

- We put `$(...)` around the command. You can then assign the output of the command.

## Conditions—String Comparisons

- All programming languages depend on *conditions* for `if` statements and for `while` loops
- Shell programming uses a built-in command which is either `test` or `[...]`
- Examples of *string* comparisons:

```
[ "$USER" = root ]      # true if the value of $USER is "root"
[ "$USER" != root ]    # true if the value of $USER is not "root"
[ -z "$USER" ]         # true if the string "$USER" has zero length
[ string1 \<; string2 ] # true if string1 sorts less than string2
[ string1 \> string2 ] # true if string1 sorts greater than string2
```

- Note that we need to quote the `>` and the `<` to avoid interpreting them as file redirection.
- *Note:* the spaces after the “[“ and before the “]” are essential.
- Also spaces are *essential* around operators

## Conditions—Integer Comparisons

- Examples of *numeric* integer comparisons:

```
[ "$x" -eq 5 ] # true if the value of $x is 5
[ "$x" -ne 5 ] # true if integer $x is not 5
[ "$x" -lt 5 ] # true if integer $x is < 5
[ "$x" -gt 5 ] # true if integer $x is > 5
[ "$x" -le 5 ] # true if integer $x is ≤ 5
[ "$x" -ge 5 ] # true if integer $x is ≥ 5
```

- Note again that the spaces after the “[“ and before the “]” are essential.
- Also spaces are *essential* around operators

## Conditions—File Tests, NOT Operator

- The shell provides many tests of information about *files*.
- Do `man test` to see the complete list.
- Some examples:

```
$ [ -f file ] # true if file is an ordinary file
$ [ ! -f file ] # true if file is NOT an ordinary file
$ [ -d file ] # true if file is a directory
$ [ -u file ] # true if file has SUID permission
$ [ -g file ] # true if file has SGID permission
$ [ -x file ] # true if file exists and is executable
$ [ -r file ] # true if file exists and is readable
$ [ -w file ] # true if file exists and is writeable
$ [ file1 -nt file2 ] # true if file1 is newer than file2
```

- *Note again:* the spaces after the “[“ and before the “]” are essential.
- Also spaces are *essential* around operators

## Conditions—Combining Comparisons

- Examples of *combining comparisons* with AND: `-a` and OR: `-o`, and *grouping* with `\(...\)`

```
# true if the value of $x is 5 AND $USER is not equal to root:
```

```
[ "$x" -eq 5 -a "$USER" != root ]
```

```
# true if the value of $x is 5 OR $USER is not equal to root:
```

```
[ "$x" -eq 5 -o "$USER" != root ]
```

```
# true if ( the value of $x is 5 OR $USER is not equal to root ) AND
```

```
# ( $y > 7 OR $HOME has the value happy )
```

```
[ \ ( "$x" -eq 5 -o "$USER" != root \) -a \
```

```
\ ( "$y" -gt 7 -o "$HOME" = happy \) ]
```

- Note again that **the spaces after the “[** and before the **”]** are **essential**.
- Do `man test` to see the information about all the operators.

## Arithmetic Assignments

- Can do with the external program `expr`
  - ...but `expr` is not so easy to use, although it is very standard and *portable*: see `man expr`
  - Easier is to use the built in `let` command
    - see `help let`
  - Examples:

```
$ let x=1+4
$ let ++x          # Now x is 6
$ let x='1 + 4'
$ let 'x = 1 + 4'
$ let x="(2 + 3) * 5" # now x is 25
$ let "x = 2 + 3 * 5" # now x is 17
$ let "x += 5"       # now x is 22
$ let "x = x + 5"    # now x is 27; NOTE NO $
```
  - Notice that you do not need to quote the special characters with `let`.
  - Quote if you want to use white space.
  - Do not put a dollar in front of variable, even on right side of assignment; see last example.

## Arithmetic Expressions with `$(...)`

- The shell interprets anything inside `$(...)` as an arithmetic expression
- You could calculate the number of days left in the year like this:

```
$ echo "There are \  
$(( (365-$(date +%j)) / 7 )) weeks \  
left till December 31"
```
- **No dollar sign** in front of variables in these arithmetic expressions.

## Arithmetic Conditions with `((...))`

- A (less portable) alternative to the arithmetic conditions in slide 27 is putting the expression in `((...))`

- So you can do

```
(( (3>2) && (4<=1) ))
```

instead of

```
[ \ ( 3 -gt 2 \) -a \ ( 4 -le 1 \) ]
```

- Operators that work with `let`, `$((...))` and `((...))` include:

```
++ -- **  
+ - * / % << >> & | ~ ! ^  
< > <= >= == !=  
? :
```

which have *exactly* the same effect as in the C programming language

- except exponentiation operator `**`, i.e., `echo $((2**20))` prints the value of  $2^{20}$ , i.e., 1048576
- For details, see

```
$ help let
```

## if Statement

- Syntax:

```
if <test-commands>
then
    <statements-if-test-commands-1-true>
elif <test-commands-2>
then
    <statements-if-test-commands-2-true>
else
    <statements-if-all-test-commands-false>
fi
```

- Example:

```
if grep nick /etc/passwd > /dev/null 2>&1
then
    echo Nick has a local account here
else
    echo Nick has no local account here
fi
```

## while Statement

- Syntax:

```
while <test-commands>
do
    <loop-body-statements>
done
```

- Example:

```
i=0
while [ "$i" -lt 10 ]
do
    echo -n "$i "    # -n suppresses newline.
    let "i = i + 1" # i=$(expr $i + 1) also works
done
```

## for Statement

- Syntax:

```
for <name> in <words>
do
    <loop-body-statements>
done
```

- Example:

```
for planet in Mercury Venus Earth Mars \
    Jupiter Saturn Uranus Neptune Pluto
do
    echo $planet
done
```

- The backslash “\” quotes the newline. It’s just a way of folding a long line in a shell script over two or more lines.



## for Loops: Another Example

- Here the shell turns `*.txt` into a list of file names ending in `".txt"`:

```
for i in *.txt
do
    echo $i
    grep 'lost treasure' $i
done
```

- You can leave the `in <words>` out; in that case, `<name>` is set to each parameter in turn:

```
i=0
for parameter
do
    let 'i = i + 1'
    echo "parameter $i is $parameter"
done
```

## for Loops: second, C-like syntax

- There is a second (less frequently used, and less portable) C-like `for` loop syntax:

```
for (( <expr1> ; <expr2> ; <expr3> ))
do
    <loop-body-statements>
done
```

- Rules: same as for arithmetic conditions—see slide [32](#)
- Example:

```
for (( i = 0; i < 10; ++i ))
do
    echo $i
done
```

## break and continue

- Use inside a loop
- Work like they do in C
- **break** terminates the innermost loop; execution goes on after the loop
- **continue** will skip the rest of the body of the loop, and resume execution on the next iteration of the loop.

## Blocks: { ... }

- A *subshell* is one way of grouping commands together, but it starts a new process, and any variable changes are localised
- An alternative is to group commands into a *block*, enclosing a set of commands in braces: { ... }
- Useful for grouping commands for *file input* or *output*
  - ... though variables are not localised
- See next slide for another application.

## Error Handling: `||`, `&&` and `exit`

- Suppose we want the user to provide exactly two parameters, and exit otherwise
- A common method of handling this is something like:

```
[ $# -eq 2 ] || { echo "Need two parameters"; exit 1; }
```

- Read this as “the number of parameters is two OR exit”
- Works because this logical OR uses short-circuit Boolean evaluation; the second statement is executed only if the first fails (is false)
- Logical AND “`&&`” can be used in the same way; the second statement will be executed only if the first is successful (true)
- A note about blocks: must have semicolon “`;`” or newline at end of last statement before closing brace

## Output: `echo` and `printf`

- To perform output, use `echo`, or for more formatting, `printf`.
- Use `echo -n` to print no newline at end.
- Just `echo` by itself prints a newline
- `printf` works the same as in the C programming language, except no parentheses or commas:

```
$ printf "%16s\t%8d\n" $my_string $my_number
```

- Do `man printf` (or look it up in the `bash` manual page) to read all about it.

## Input: the `read` Command

- For input, use the built-in shell command `read`
- `read` reads standard input and puts the result into one or more variables
- If use one variable, variable holds the whole line
- Syntax:

```
read <var1>...
```

- Often used with a `while` loop like this:

```
while read var1 var2
do
    # do something with $var1 and $var2
done
```

- Loop terminates when reach end of file
- To prompt and read a value from a user, you could do:

```
while [ -z "$value" ]; do
    echo -n "Enter a value: "
    read value
done
# Now do something with $value
```

## set: Splitting a Multi-Word Variable

- Sometimes may want to split a multi-word variable into single-word variables
- `read` won't work like this:

```
MY_FILE_INFO=$(ls -lR | grep $file)
# ...
echo $MY_FILE_INFO | read perms links \
user group size month day time filename
```

- Use the builtin command `set` instead:

```
MY_FILE_INFO=$(ls -lR | grep $file)
# ...
set $MY_FILE_INFO
perms=$1 links=$2 user=$3 group=$4 size=$5
month=$6 day=$7 time=$8 filename=$9
```

## More about set, and IFS

- `set` splits its arguments into pieces (usually) at whitespace
- It sets the first value as `$1`, the second as `$2`, and so on.
- Note that you can change how `set` and the shell splits things up by changing the value of a special variable called `IFS`
- `IFS` stands for *Internal Field Separator*
- Normally the value of `IFS` is the string “`<space><tab><newline>`”
- Next slide shows how changing `IFS` to a colon let us easily split the `PATH` into separate directories:

~> next slide

## Example: Changing IFS

- Notice that here, I make the change to IFS in a subshell. I have simply typed the loop at the prompt.
- As I said in slide 17, changes in a subshell are local to the subshell:

```
$ echo $PATH
/usr/bin:/bin:/usr/X11R6/bin:/home/nicku/bin
$ (IFS=
> for dir in $PATH
> do
>     echo $dir
> done
> )
/usr/bin
/bin
/usr/X11R6/bin
/home/nicku/bin
```

## case Statement

- Similar to the `switch` statement in C, but more useful and more general
- Uses pattern matching against a string to decide on an action to take
- Syntax:

```
case <expression> in
    <pattern1> )
        <statements> ;;
    <pattern2> )
        <statements> ;;
    ...
esac
```

## case Statement: Example

- This example code runs the appropriate program on a graphics file, depending on the file extension, to convert the file to another format:

```
case $filename in
  *.tif)
    tifftopnm $filename > $ppmfile
    ;;
  *.jpg)
    tjpeg $filename > $ppmfile
    ;;
  *)
    echo -n "Sorry, cannot handle this "
    echo "graphics format"
    ;;
esac
```

## **shift:** Move all Parameters Up

- Sometimes we want to process command-line parameters in a loop
- The **shift** statement is made for this
- Say that we have four parameters:

<u>parameter</u>	<u>value</u>	<u>parameter</u>	<u>value</u>
\$1	one	\$3	three
\$2	two	\$4	four

- Then after executing the **shift** statement, the values are now:

<u>parameter</u>	<u>value</u>	<u>parameter</u>	<u>value</u>
\$1	two	\$3	four
\$2	three	\$4	<i>no longer exists</i>



## shift: Many Places

- You can give a number argument to **shift**:
  - If before, we have four parameters:

<u>parameter</u>	<u>value</u>	<u>parameter</u>	<u>value</u>
\$1	one	\$3	three
\$2	two	\$4	four

- After executing the statement:

```
$ shift 2
```

we have two parameters left:

<u>parameter</u>	<u>value</u>	<u>parameter</u>	<u>value</u>
\$1	three	\$3	<i>no longer exists</i>
\$2	four	\$4	<i>no longer exists</i>

## Command-Line Options—1

- Sometimes we want to modify the behaviour of a shell script
  - For example, want an *option* to show more information on request
  - could use an option “-v” (for “verbose”) to tell the shell script that we want it to tell us more information about what it is doing
  - If script is called **showme**, then we could use our **-v** option like this:

```
$ showme -v
```
  - the script then shows more information.

## Command-Line Options—2

- For example, We might provide an option to give a starting point for a script to search for SUID programs
- Could make the option `-d` *<directory>*
- If script is called `findsuid`, could call it like this:

```
$ findsuid -d /usr
```

to tell the script to start searching in the directory `/usr` instead of the current directory

## Command-Line Options—3

- We could do this using `shift`, a `while` loop, and a `case` statement, like this:

```
while [ -n "$(echo $1 | grep '-')" ]
do
    case $1 in
        -v) VERBOSE=1 ;;
        -d)
            shift
            DIRECTORY=$1
            ;;
        *) echo "usage: $0 [-v] [-d dir]"
           exit 1 ;;
    esac
    shift
done
```

## getopts: Command-Line Options—4

- Problems with above solution: inflexibility:
  - Does not allow options to be “bundled” together like `-abc` instead of `-a -b -c`
  - Requires a space between option and its argument, i.e., doesn’t let you do `-d/etc` as well as `-d /etc`
  - Better method: use the built-in command `getopts`:

```
while getopts ":vd:" opt
do
    case opt in
        v) VERBOSE=1 ;;
        d) DIRECTORY=$OPTARG ;;
        *) echo "usage: $0 [-v] [-d dir]"
           exit 1 ;;
    esac
done
shift $((OPTIND - 1))
```

## getopts: Command-Line Options—5

- `getopts` takes two arguments:
  - first comes the string that can contain letters and colons.
    - Each letter represents one option
    - A colon comes after a letter to indicate that option takes an argument, like `-d directory`
    - A colon at the beginning makes `getopts` less noisy, so you can provide your own error message, as shown in the example.
  - The second is a variable that will hold the option (without the hyphen “-”)
- Shift out all processed options using the variable `OPTIND`, leaving any other arguments accessible
- Search for `getopts` in the `bash` man page

## Temporary Files: `mktemp`

- Sometimes it is convenient to store temporary data in a temporary file
- The `mktemp` program is designed for this
- We use it something like this:

```
TMPFILE=$(mktemp /tmp/temp.XXXXXX) || exit 1
```
- `mktemp` will create a new file, replacing the “XXXXXX” with a random string
- Do `man mktemp` for the complete manual.

## Signals that may Terminate your Script

- Many key strokes will send a *signal* to a process
- Examples:
  - `Control-C` sends a **SIGINT** signal to the current process running in the foreground
  - `Control-\` sends a **SIGQUIT** signal
- When you log out, all your processes are sent a **SIGHUP** (hangup) signal
- If your script is connected to another process that terminates unexpectedly, it will receive a **SIGPIPE** signal
- If anyone terminates the program with the **kill** program, the default signal is **SIGTERM**

## Signals: trap

- Sometimes you want your script to clean up after itself nicely, and remove temporary files
- Do this using **trap**

## Signals: trap Example

- Suppose your script creates some temporary files, and you want to remove them if your script receives any of these signals
- You can “catch” the signal, and remove the files when the signals are received before the program terminates
- Suppose the temporary files have names stored in the variables `TEMP1` and `TEMP2`
- Then you would trap these signals like this:

```
trap "rm $TEMP1 $TEMP2" HUP INT QUIT PIPE TERM
```

- Conveniently, (but not very portably), `bash` provides a “pretend” signal called `EXIT`; can add this to the list of signals you trap, so that the temporary files will be removed when the program exits normally.

## Functions

- The shell supports functions and function calls
- A function works like an external command, except that it does not start another process
- Syntax:

```
function <funcname>
{
    <shell commands>
}
```

Or:

```
<funcname> ()
{
    <shell commands>
}
```

## Parameters in Functions

- Work the same as parameters to entire shell script
- First parameter is \$1, second is \$2,..., the tenth parameter is \${10}, and so on.
- \$# is the number of parameters passed to the function
- As with command line parameters, they are read-only
- Assign to meaningful names to make your program more understandable

## Example, Calling a Function

- This is a simple example program:

```
#!/bin/sh

function cube {
    echo $(( $1 * $1 * $1 ))
}

j=$(cube 5)
echo $j      # Output is 125
```

- Note the use of *command substitution to get a return value*
- The *function prints result to standard output*.

## Debugging Shell Scripts—1

- If you run the script with:

```
$ sh -v <script>
```

then each statement will be printed as it is executed

- If you run the script with:

```
$ sh -x <script>
```

then an execution trace will show the value of all variables as the script executes.



## Debugging Shell Scripts—2

- Use `echo` to display the value of variables as the program executes
- You can turn the `-x` shell option on in any part of your script with the line:

```
set -x
```

and turn it *off* with:

```
set +x
```

- No, that's not a typo: `+x` turns it *off*, `-x` turns it *on*.
- The book *Learning the bash Shell* includes a `bash` shell debugger if you get desperate

## Writing Shell Scripts

- Build your shell script *incrementally*:
  - Open the `editor` in one window (*and leave it open*), have a terminal window open in which to **run your program as you write it**
  - *Test as you implement*: this makes shell script development *easy*
  - Do *not* write a very complex script, and *then* begin testing it!
- Use the standard software engineering practice you know:
  - Use *meaningful* variable names, function names
  - Make your program *self-documenting*
  - Add *comment blocks* to explain obscure or difficult parts of your program

## Useful External Programs—1

Each of these has a manual page, and many have info manuals. Read their online documentation for more information.

- **awk** — powerful tool for processing columns of data
- **basename** — remove directory and (optionally) extension from file name
- **cat** — copy to standard output
- **cut** — process columns of data
- **du** — show disk space used by directories and files
- **egrep, grep** — find lines containing patterns in files
- **find** — find files using many criteria

## Useful External Programs—2

- **last** — show the last time a user was logged in
- **lastb** — show last bad log in attempt by a user
- **rpm** — RPM package manager: manage software package database
- **sed** — stream editor: edit files automatically
- **sort** — sort lines of files by many different criteria
- **tr** — translate one set of characters to another set
- **uniq** — replace repeated lines with just one line, optionally with a count of the number of repeated lines

## Regular Expressions

- Many programs and programming languages use *regular expressions*, including **Java** 1.4 and later, **Perl**, **VB.NET**, **C#** (and any language using the .NET Framework), **PHP**, **Python**, **Ruby**, **Tcl** and **MySQL** (plus many others; even **MS Word** uses regular expressions under Edit → Find → More → Use wildcards)
- These programs use regular expressions:
  - **grep**, **egrep**, **sed**, **awk**
- All programmer's editors support regular expressions (**Emacs**, **vi**, ...)
- *Regular expressions* provide a *powerful language* for *manipulating data* and *extracting important information* from masses of data

## What is In a Regular Expression?

- There are two types of character in a regular expression:
  - *Metacharacters*
    - These include:
    - **\* \ . + ? ^ ( ) [ { |**
  - Ordinary, *literal* characters:
    - i.e., all the other characters that are not metacharacters

## Literal characters

- Find all lines containing "chan" in the password file:

```
$ grep chan /etc/passwd
```

- The *regular expression* is "chan"
- It is made entirely of literal characters
- It matches only lines that contain the *exact* string
- It will match lines containing the words **chan**, **changed**, **merchant**, **mechanism**,...

## Character Classes: [...]

- A character class represents *one* character
- Examples:

```
# Find all words in the dictionary that contain a vowel:  
$ grep "[aeiou]" /usr/share/dict/words  
# Find all lines that contain a digit:  
$ grep "[0123456789]" /usr/share/dict/words  
# Find all lines that contain a digit:  
$ grep "[0-9]" /usr/share/dict/words  
# Find all lines that contain a capital letter:  
$ grep "[A-Z]" /usr/share/dict/words
```

## Negated Character Classes: [^...]

- Examples of negated character classes:

```
# Find all words in the dictionary
# that contain a character that is not a vowel:
$ grep "[^aeiou]" /usr/share/dict/words
# Two ways of finding all lines that contain
# a character that is not a digit:
$ grep "[^0123456789]" /usr/share/dict/words
$ grep "[^0-9]" /usr/share/dict/words
# Find all lines that contain a character
# that is not a digit, or a letter
$ grep "[^0-9a-zA-Z]" /usr/share/dict/words
```

- Remember: each set of square brackets represents exactly *one* character.

## Match Any Character

- The dot “.” matches *any single character*, except a newline.
- The pattern ‘.....’ matches all lines that contain at least five characters

## Matching the Beginning or End of Line

- To match a line that contains exactly five characters:

```
$ grep '^.....$' /usr/share/dict/words
```

- The hat, `^` represents the *position* right at the *start* of the line
- The dollar `$` represents the *position* right at the *end* of the line.
- Neither `^` nor `$` represents a character
- They represent a position
- Sometimes called *anchors*, since they anchor the other characters to a specific part of the string

## Match Repetitions: `*`, `?`, `+`, `{n}`, `{n,m}`

- To match *zero* or more:
- `a*` represents zero or more of the lower case letter `a`, so the pattern will match "" (the empty string), "a", "aa", "aaaaaaaaaaaaaaaa", "qwewtrrryu" or the "nothing" in front of any string!
- To match *one* or more:
- `'a+'` matches one or more "a"s
- `'a?'` matches zero or one "a"
- `'a{10}'` matches exactly 10 "a"s
- `'a{5,10}'` matches between 5 and 10 (inclusive) "a"s

## Matching Alternatives: “|”

- the vertical bar represents alternatives:
- The regular expression `'nick|albert|alex'` will match either the string “nick” or the string “albert” or the string “alex”
- Note that the vertical bar has very low precedence:
- the pattern `'^fred|nurk'` matches “fred” only if it occurs at the start of the line, while it will match “nurk” at any position in the line

## Putting it All Together: Examples

- Find all words that contain at least three ‘a’s:

```
$ egrep 'a.*a.*a' /usr/share/dict/words
```

- Why is this different from

```
$ egrep 'aaa' /usr/share/dict/words
```

- Find all words that begin in ‘a’ and finish in ‘z’, ignoring case:

```
$ egrep -i '^a.*z$' /usr/share/dict/words
```

- How is this different from:

```
$ egrep -i '^a.*z' /usr/share/dict/words
```

- Find all words that contain at least two vowels:

```
$ grep '[aeiou].*[aeiou]' /usr/share/dict/words
```

- Find all words that contain *exactly* two vowels:

```
$ egrep \  
'^[^aeiou]*[aeiou][^aeiou]*[aeiou][^aeiou]*$' \  
/usr/share/dict/words
```

- Find all lines that are empty, or contain only spaces:

```
$ grep '^ *$' file
```



## Basic awk

- **awk** is a complete programming language
- Mostly used for one-line solutions to problems of *extracting columns of data* from text, and processing it
- A complete book is available on **awk**; you can buy it here: <http://www.oreilly.com/catalog/awkprog3/> or
- read it on your computer, as it is the official manual for **gawk** (GNU awk); do

```
$ info gawk
```

or read it in Emacs.

- A printable postscript file of the book (353 pages) is on my computer at `/usr/share/doc/gawk-3.1.3/gawk.ps`

## What Does awk Do?

- **awk** reads file(s) or standard input *one line at a time*, and
- automatically *splits the line into fields*, and calls them `$1`, `$2`, ..., `$NF`
- `NF` is equal to the *number of fields* the line was split into
- `$0` contains the *whole line*
- **awk** has an option `-F` that allows you to select another pattern as the *field separator*
  - Normally **awk** splits *columns* by *white space*
- To execute code after all lines are processed, create an *END block*.

## awk Examples

- Print the sizes of all files in current directory:

```
ls -l | awk '{print $5}'
```

- Add the sizes of all files in current directory:

```
ls -l | awk '{sum += $5} END{print sum}'
```

- Print only the permissions, user, group and file names of files in current directory:

```
ls -l | awk '{print $1, $3, $4, $NF}'
```

## sed—the Stream Editor

- `sed` provides many facilities for editing files
- The *substitute* command, `s///`, is the most important
- The syntax (using `sed` as an editor of standard input), is:

```
$ sed 's/<original>/<replacement>/'
```

- Example: replace the first instance of `Windows` with `Linux` on each line of the input:

```
sed 's/Windows/Linux/'
```

- Example: replace *all* instances of `Windows` with `Linux` on each line of the input:

```
sed 's/Windows/Linux/g'
```

- Note: by default, `sed` uses “basic regular expressions”, which require a backslash ‘\’ in front of the metacharacters ‘{’, ‘(’, ‘)’, ‘|’, ‘+’ and ‘?’.
- To use “extended regular expressions” (which we covered here), call `sed` with the option `-r`, as in this example:

```
$ sed -r s/a+//
```

## sed—Backreferences

- You can match part of the *<original>* in a `sed -r` substitute command, and put that part back into the replacement part.
- You enclose the part you want to refer to later in `(...)`
- You can get the first value in the replacement part by `\1`, the second opening parenthesis of `(...)` by `\2`, and so on.

## sed—Backreferences: Example

- If you do

```
$ find /etc | xargs file -b
```

you will get a lot of output like this:

```
symbolic link to bg5ps.conf.zh_TW.Big5
symbolic link to rc.d/rc.local
symbolic link to rc.d/rc
symbolic link to rc.d/rc.sysinit
symbolic link to ../../X11/xdm/Xservers
```

- If you want to edit each line to remove everything after “symbolic link”, then you could pipe the data through `sed` like this:

```
$ find /etc | xargs file -b \
| sed -r 's/(symbolic link).*/\1/'
```

- See slide [83](#) for an application

## find Examples

- Count the number of unique manual pages on the computer:

```
$ find /usr/share/man -type f | wc -l
```

- Print a table of types of file under the `/etc` directory, with the most common file type down at the bottom:

```
$ find /etc | xargs file -b \
| sed -r 's/(symbolic link).*/\1/' \
| sort \
| uniq -c \
| sort -n
```

## Finding SUID Programs

- Finding SUID or SGID files:

```
$ sudo find / -type f \  
\( perm -2000 -o -perm -4000 \) \  
> files.secure
```

- Let's compare with a list of SUID and SGID files to see if there are any changes, since SUID and SGID programs can be a *security risk*:

```
$ sudo find / -type f \  
\( perm -2000 -o -perm -4000 \) \  
| diff - files.secure
```

## A find Example with Many Options

- Set all directories to have the access mode 771, set all backup files (\*.BAK) to mode 600, all shell scripts (\*.sh) to mode 755, and all text files (\*.txt) to mode 644:

```
$ find . \  
  \(-type d -a exec chmod 771 {} \; \) -o \  
  \(-name "*.BAK" -a exec chmod 600 {} \; \) -o \  
  \(-name "*.sh" -a exec chmod 755 {} \; \) -o \  
  \(-name "*.txt" -a exec chmod 644 {} \; \)
```

## rpm Database Query Commands

- The `rpm` software package management system includes a database with very detailed information about every file of every software package that is installed on the computer.
- You can query this database using the `rpm` command.
- The manual page does not give the complete picture, but there is a book called *Maximum RPM* that comes on the Red Hat documentation CD
- This package is installed on `ictlab`
- You can see the appropriate section at this URL:

<http://nicku.org/doc/maximum-rpm-1.0/html/s1-rpm-query-parts.html>