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.

---

**Shell Programming—an Introduction**

Copyright Conditions: Open Publication License (see [http://www.opencontent.org/oopml/](http://www.opencontent.org/oopml/))

Nick Urbanik

nick@nicks.org

A computing department

---

**Aim**

---

**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


There is a free on-line book about shell programming at:
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

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.

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
<th>See slide</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>Home directory</td>
<td>§ 7</td>
</tr>
<tr>
<td>*</td>
<td>Command substitution, Better: $(...)</td>
<td>§ 24</td>
</tr>
<tr>
<td>#</td>
<td>Comment</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>Variable expression</td>
<td>§ 15</td>
</tr>
<tr>
<td>&amp;</td>
<td>Background Job</td>
<td>2.10 on page 41</td>
</tr>
<tr>
<td>*</td>
<td>File name matching wildcard</td>
<td>2.18 on page 49</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>Start subshell</td>
<td>45, 17, 39</td>
</tr>
<tr>
<td>)</td>
<td>End subshell</td>
<td></td>
</tr>
<tr>
<td>[</td>
<td>Start character set file name matching</td>
<td>2.9 on page 40</td>
</tr>
<tr>
<td>]</td>
<td>End character set file name matching</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>Start command block</td>
<td>39</td>
</tr>
<tr>
<td>;</td>
<td>Command separator</td>
<td>40</td>
</tr>
<tr>
<td>\</td>
<td>Quote next character</td>
<td>23</td>
</tr>
<tr>
<td>'</td>
<td>Strong quote</td>
<td></td>
</tr>
<tr>
<td>&quot;</td>
<td>Weak quote</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>Input redirect</td>
<td>2.7 on page 38</td>
</tr>
<tr>
<td>&gt;</td>
<td>Output redirect</td>
<td>2.6 on page 37</td>
</tr>
<tr>
<td>/</td>
<td>Pathname directory separator</td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>Single-character match in filenames</td>
<td>2.18 on page 49</td>
</tr>
<tr>
<td>!</td>
<td>Pipeline logical NOT</td>
<td>28</td>
</tr>
<tr>
<td>⟨</td>
<td>shell normally splits at white space</td>
<td></td>
</tr>
</tbody>
</table>

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”.

Special Characters—continued: 2

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
<th>See slide</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>Start subshell</td>
<td>§ 39</td>
</tr>
<tr>
<td>)</td>
<td>End subshell</td>
<td>§ 40</td>
</tr>
<tr>
<td>[</td>
<td>Start character set file name matching</td>
<td>§ 49</td>
</tr>
<tr>
<td>]</td>
<td>End character set file name matching</td>
<td>§ 49</td>
</tr>
<tr>
<td>{</td>
<td>Start command block</td>
<td>§ 40</td>
</tr>
<tr>
<td>;</td>
<td>Command separator</td>
<td>§ 40</td>
</tr>
<tr>
<td>\</td>
<td>Quote next character</td>
<td>§ 23</td>
</tr>
<tr>
<td>'</td>
<td>Strong quote</td>
<td>§ 23</td>
</tr>
<tr>
<td>&quot;</td>
<td>Weak quote</td>
<td>§ 23</td>
</tr>
</tbody>
</table>

Special Characters—continued: 3

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

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

```bash
$ 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:

```bash
$ find . -name \*.jpg
$ locate '/usr/bin/c*'
$ grep 'main.*('*.c
```

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:

```bash
$ x=10 # correct
$ x = 10 # wrong: try to execute program called "x"
```

Read value of variable:
- put a ‘$’ in front of variable name

Example:

```bash
$ echo "The value of x is $x"
```

Variables—Assignments

You can put multiple assignments on one line:

```bash
i=0 j=10 k=100
```

You can set a variable temporarily while executing a program:

```bash
$ 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:

```bash
$ 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:

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

You can make it disappear totally using `unset`:

```bash
$ 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:

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

<table>
<thead>
<tr>
<th>variable</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>shell-script</td>
</tr>
<tr>
<td>$1</td>
<td>param1</td>
</tr>
<tr>
<td>$2</td>
<td>param2</td>
</tr>
<tr>
<td>$3</td>
<td>param3</td>
</tr>
<tr>
<td>$4</td>
<td>param4</td>
</tr>
<tr>
<td>$#</td>
<td>number of parameters to the program, e.g., 4</td>
</tr>
</tbody>
</table>

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:
```bash
#!/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 $$
```

Braces and Parameters after $9

- Need braces to access parameters after $9:
  ```bash
  $ cat paramten
  #!/bin/sh
echo $10
echo $11
  $ . /paramten a b c d e f g h i j
  a0
  
  Notice that $10 is the same as ${10}, i.e., the first parameter "a" then the literal character zero "0"
  ```

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

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:
```bash
[ "$USER" = root ] # true if the value of $USER is "root"
[ "$USER" ~ root ] # true if the value of $USER is not "root"
[ -s "$USER" ] # true if the string "$USER" has been length
[ strings !~ strings ] # true if string1 sorts less than string2
[ strings !< strings ] # 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

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:
  ```bash
  $ test=123
  $ echo $test
  123
  $$
  ```

More about Quoting

- Double quotes: "..." stop the special behaviour of all special characters, except for:
  ```bash
  variable interpretation ($)  
  backticks (') — see slide [24]  
  backslash (\)
  ```
- Single quotes ‘...’:
  ```bash
  stop the special behaviour of all special characters
  ```

Backslash:

- Preserves literal behaviour of character, except for newline; see slides [23][25][25]
- 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.

Conditions—Integer Comparisons

- Examples of numeric integer comparisons:
  ```bash
  [ "$a" -eq 5 ]  # true if the value of $a is 5
  [ "$a" -ne 5 ]  # true if integer $a is not 5
  [ "$a" -lt 5 ]  # true if integer $a is less than 5
  [ "$a" -le 5 ]  # true if integer $a is less than or equal to 5
  [ "$a" -gt 5 ]  # true if integer $a is greater than 5
  [ "$a" -ge 5 ]  # true if integer $a is greater than or equal to 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
- `[ -o file ]` # true if file has SUID permission
- `[ -g file ]` # true if file has SUID 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 writable
- `[ [ file -nt file2 ] ]` # true if file is newer than file2

Note: the spaces after the `[]` and before the `[]` are essential.

Also spaces are essential around operators

Arithmetic Assignments

Can do with the external program `expr`;

- `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

- `let` helps:

  Examples:
  - `let x=1+4`
  - `let ++x` # Now x is 6
  - `let x=x + 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 *= 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 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:

```
++ -- **
+ - / % << >> & | ˜ ! ˆ
```

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.,

```
2131733470191873393
```

if Statement

- Syntax:

```bash
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
```

for Statement

- Syntax:

```bash
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":

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

You can leave the in (work) out; in that case, (name) is set to each parameter in turn:

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

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:
  ```bash
  [ $# -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:
  ```bash
  printf "%16s	%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:
```bash
read {var1}...
```

- Often used with a `while` loop like this:
  ```bash
  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:
  ```bash
  while [ -z "$value" ]; do
    echo -n "Enter a value: ";
  done
  ```

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:
  ```bash
  MY_FILE_INFO=$(ls -lR | grep $file)
  ```
- `echo $MY_FILE_INFO | read perms links user group size month day time filename`
- Use the `builtin command set` instead:
  ```bash
  MY_FILE_INFO=$(ls -lR | grep $file)
  ```
- `set $MY_FILE_INFO`
  ```bash
  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:

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

Example: Changing IFS

- Notice that here, I make the change to `IFS` in a subshell.
- As I said in slide 17, changes in a subshell are local to the subshell:

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

```bash
case $filename in
  *.tif)
tifftopnm $filename > $ppmfile
  ;;
  *.jpg)
tjpeg $filename > $ppmfile
  ;;
  *)
echo "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:

<table>
<thead>
<tr>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1</td>
<td>one</td>
</tr>
<tr>
<td>$2</td>
<td>two</td>
</tr>
<tr>
<td>$3</td>
<td>three</td>
</tr>
<tr>
<td>$4</td>
<td>four</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1</td>
<td>two</td>
</tr>
<tr>
<td>$2</td>
<td>three</td>
</tr>
<tr>
<td>$3</td>
<td>four</td>
</tr>
</tbody>
</table>

**shift: Many Places**

- You can give a number argument to `shift`:
  
  ```bash
  $ shift 2
  ```

- After executing the statement:
  
<table>
<thead>
<tr>
<th>parameter</th>
<th>value</th>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1</td>
<td>three</td>
<td>$3</td>
<td>no longer exists</td>
</tr>
<tr>
<td>$2</td>
<td>four</td>
<td>$4</td>
<td>no longer exists</td>
</tr>
</tbody>
</table>

**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:
  
  ```bash
  $ 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:

  ```bash
  $ 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:

```bash
while [ -n "$1" ]
do
case $1 in
  -v) VERBOSITY=1 ;;
  -d) DIRECTORY=$OPTARG ;;
  *) 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`:

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

Functions

A function works like an external command, except that it does not start another process.

Syntax:

```
function (functname)
{
  (shell commands)
}
Or:
(functname) ()
{
  (shell commands)
}
```

Signals that may Terminate your Script

Many key strokes will send a signal to a process.

Examples:
- `Ctrl-C` sends a `SIGINT` signal to the current process running in the foreground
- `Ctrl-D` 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 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:

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

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

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:

```bash
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: trap

Sometimes you want your script to clean up after itself nicely, and remove temporary files.

Do this using `trap`.

Functions
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:

```bash
#!/bin/sh

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

ej=${cube 5}
echo $j # Output is 125
```

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

Useful External Programs—2

- `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

Regular Expressions
**What is In a Regular Expression?**

There are two types of character in a regular expression:

- **Metacharacters**
  - These include:
    - `^ \ . ? * + ? ^ ( ) \ [ { |` i.e., all the other characters that are not metacharacters

- **Literal characters**
  - 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 "[0-9]" /usr/share/dict/words`

- **Examples of negated character classes:**
  - # Find all lines that contain a capital letter: 
    - `grep "[^A-Z]" /usr/share/dict/words`
  - # Find all lines that contain a digit: 
    - `grep "[^0-9]" /usr/share/dict/words`

### 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 `^

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", "aaaaaaaaaaaaa", "qwewtrrry" 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:
$ egrep '\[aeiou]+\[aeiou]\[aeiou]+\[aeiou]+$' /usr/share/dict/words

Find all words that contain exactly two vowels:
$ egrep '/\[aeiou]\\[aeiou]\\[aeiou]\\[aeiou]+$/usr/share/dict/words

Find all lines that are emty or contain only spaces:
$ ls -l | awk '{$1=$2,...,$NF}'

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

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—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.

If you do
$ find /etc | xargs file -b
you will get a lot of output like this:
symbolic link to bq5ps.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 53 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
 '\( perm -2000 -o -perm -4000 \) \n > 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
 '\( perm -2000 -o -perm -4000 \) \n | 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 \n \{ -name "*.BAK" -a exec chmod 600 \{{}\} \o \n \{ -name "*.sh" -a exec chmod 755 \{{}\} \o \n \{ -name "*.txt" -a exec chmod 644 \{{}\} \o
```

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