Chapter 8
Shell Scripting

Chapter 8 Outline

- In this chapter we will learn how to:
  - ✓ Create shell scripts
  - ✓ Use variables and environment variables
  - ✓ Pass parameters to shell scripts
  - ✓ Use branches and loops within scripts
  - ✓ Define aliases
  - ✓ Use shell startup files

Basic shell scripting

- Basic shell scripting
  - Introducing shell scripts
  - The UNIX shells
  - Creating a script
  - Variables
  - Environment variables
  - Inheritance of variables
  - Important internal bash variables
  - Passing parameters to scripts
  - Command substitution

Introducing shell scripts

- A *shell script* is a sequence of shell commands stored in a file
  - Script can be run just by entering its name
  - Useful for automating system administration and other tasks
- Easy to write
  - Build on commands that are already familiar from command-line use
  - Can be developed incrementally
- Portable
  - Scripts are not compiled into machine code, they are interpreted
  - Work across multiple platforms and architectures
- Slow
  - Not good for computationally intensive tasks
  - A fully compiled language such as C will run much faster

The UNIX shells

- Several shells have been written for UNIX and linux. They have a substantial set of core features in common, but the syntax of some constructs differs.
  - This chapter focuses on `bash`
- Bourne shell (`/bin/sh`)
  - The original shell written by Steve Bourne
- Bourne again shell (`/bin/bash`)
  - Extends `/bin/sh` and is backward compatible with it
  - The standard shell on Linux
- C shell (`/bin/csh`)
  - A shell with a syntax (slightly) resembling the C language
- Korn shell (`/bin/ksh`)
  - Also popular particularly on mainstream UNIX systems

Creating a script

- Scripts can be created with any text editor
- The file `greet.sh`:

```
#!/bin/bash
# A simple script
echo "Hello world!"
exit 0
```

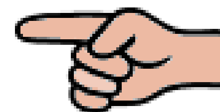
Specifies the interpreter for this script

A comment

The exit status of the script

- A script can be executed by explicitly starting a shell with the script name as an argument.

```
$ bash greet.sh
Hello World!
```



Do now!

Create and run this script

ict@innovation

FOSSFA

Creating a script (continued)

- If the script has execute permission it can be run by using its name as a command
  - A new shell (subshell) is started implicitly

```
$ greet.sh
bash: greet.sh: command not found
$ ./greet.sh
bash: ./greet.sh: Permission denied
$ chmod u+x greet.sh
$ ./greet.sh
Hello World!
```

The current directory is not on our search path

The script is not executable

Now it works!

- It is conventional (but not essential) to use the filename suffix `.sh`
  - Makes it clear that this is a shell script

Variables

- Like other programming languages, the shell has *variables*
- Variables are defined with the syntax 'NAME=value' and referenced with the syntax $NAME
- Example:

```
$ GREETING="Good morning"
$ echo $GREETING
Good morning
```

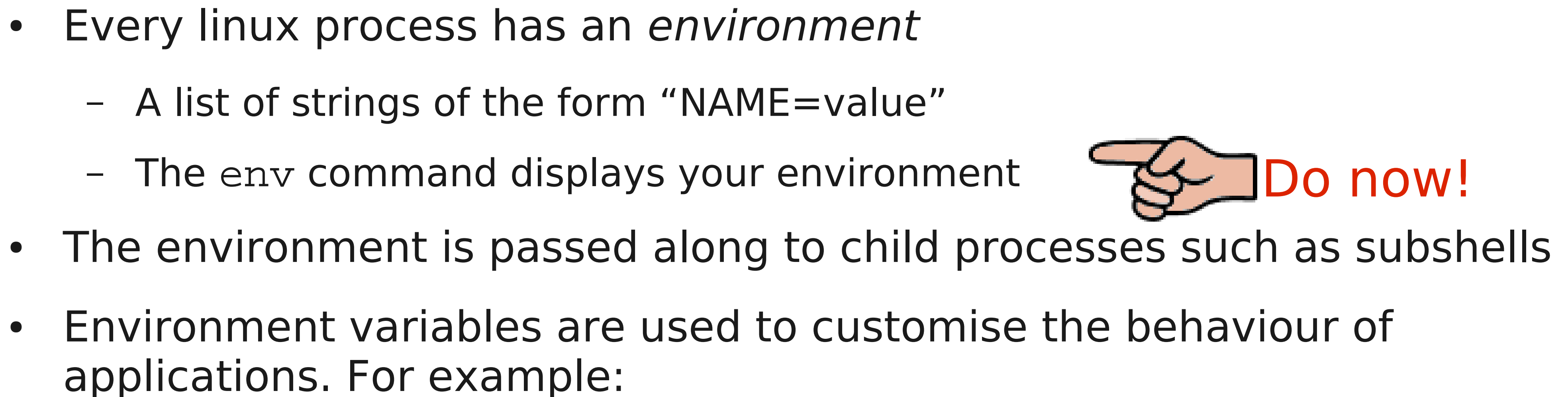
- Some variables are used to customise the behaviour of the shell, for example:

HISTSIZE	Number of commands to keep in the history list
PS1	The 'primary prompt' string
UID	Numeric user ID of the logged-in user

- Variables are local to the shell and are not inherited by subshells

Environment variables

- Every linux process has an *environment*
  - A list of strings of the form “NAME=value”
  - The `env` command displays your environment



- The environment is passed along to child processes such as subshells
- Environment variables are used to customise the behaviour of applications. For example:

`HOME`	Home directory (where 'cd' takes you with no arguments)
`PATH`	The list of directories the shell will look in for commands
`DISPLAY`	Which X server a graphical application will connect to
`PWD`	Pathname of your current directory

- The `export` command puts an existing variable into the environment

```
$ export GREETING
```

- You can define a variable and export it in a single command:

```
$ export COLOUR=”yellow”
```

Inheritance of variables

- Local shell variables are not inherited by subshells, environment variable are:

```
$ GREETING="Good morning"
$ export COLOUR="yellow"
$ bash
$ echo $GREETING

$ echo $COLOUR
yellow
$ exit
$
```

Define a local variable

Define another local variable and put it into the environment

Start a subshell

The local variable is not inherited

The environment variable is inherited

Quit from the subshell

This prompt is from the parent shell

Important internal `bash` variables

$IFS	Input field separator – the character that separates arguments
$PS1	The primary shell prompt
$PS2	The secondary shell prompt
$?	The exit status (return value) of the most recent child process – 0 means true (success), nonzero means false (failure)
$$	The process ID of the current shell
$#	The number of arguments passed to the shell
$0- $9	The positional parameters passed to the script – $0 is the command name, $1 is the first argument, etc
$*	All the positional parameters (except $0)

Passing parameters to scripts

- A script can access its command line arguments using `$1, $2, ...`
- The script `printargs.sh`

```
#!/bin/bash
echo There are $# arguments
echo The first two are $1 and $2
```

```
$ ./printargs.sh apple orange banana
There are 3 arguments
The first two are apple and orange
$
```

Command substitution

- The notation `$(somecommand)` runs the program `somecommand` and substitutes its standard output back into the command line
  - Often used to generate arguments for some other command, or to set the value of a variable

```
$ echo it is now $(date)
it is now Sat Dec 6 21:30:30 GMT 2003
$ xclock -title $(hostname) &
[1] 2662
```

Display a clock with the hostname in the title bar (The & at the end of the line runs the command in the background)

```
$ NPROCS=$(ps ax | wc -l)
$ echo there are $NPROCS processes
there are 74 processes
```

Count the number of processes currently running

```
$ less $(grep -l README *)
```

View all files that contain the string 'README'

Control Structures

- Control structures
  - Reading from standard input
  - Exit status
  - Branching with `if`
  - The `test` command
  - A script to compare dates
  - Processing a list with a `for` loop
  - Example of a `for` loop
  - Generating the list for a `for` loop

Reading from standard input

- The `read` command will read a line from `stdin` and store it in a variable

```
#!/bin/bash
echo “Please enter a value”
read THEVALUE
echo $THEVALUE
```

- **Exercise**: Modify the above script to prompt for and read in the user's first name and last name. (Prompt for and read each name separately and store the names in two separate variables.) Then print a greeting using the user's full name

Exit status

- Every program that runs returns an exit status to its parent process
  - Zero indicates normal termination ("success")
  - Non-zero indicates abnormal termination ("failure")
- The exit status of the most recent command is available in the special variable $?

```
$ cp /etc/fstab .
$ echo $?
0
$ cat /etc/shadow
cat: /etc/shadow: Permission denied
$ echo $?
1
$ grep -q root /etc/passwd
$ echo $?
0
$ grep -q xyzzy /etc/passwd
$ echo $?
1
```

Success

Failure

Success

Failure

Branching with `if`

- The keywords `if`, `then`, `else` and `fi` allow the construction of branches

```
if test-command
then
   command1
else
   command2
fi
```

`test-command` is run and its exit status is tested

If the exit status is true (zero) these commands are executed

If the exit status is false (nonzero) these commands are executed

The `else` part is optional

Branching with if (continued)

- Recall that `grep` searches files for lines matching a text pattern
  - It returns 0 exit status if it finds a match, 1 if it doesn't
  - It normally outputs the matching lines to `stdout`, `-q` supresses this

```
#!/bin/bash
if grep -q $1 /etc/passwd
then
    echo The user has an account
else
    echo The user does not have an account
fi
```

The script checkuser.sh

```
$ ./checkuser.sh root
The user has an account
$ ./checkuser.sh xyzzy
The user does not have an account
```

The `test` command

- The command `test` is often used as the test-command for an `if` test
  - It returns true or false exit status
- File property tests include:

`test -e xyzzy`	Does the file exist?
`test -d /etc`	Is the file a directory?
`test -x greet.sh`	Is the file executable?

- Integer comparison tests include:

`test $A -eq 10`	Is A equal to 10?
`test $X -gt 5`	Is X greater than 5?

- String comparison tests include:

`test $NAME = Tux`	Are the two strings equal?

A script to compare dates

- We will implement a script that compares two dates
  - The idea is for the user to enter his date of birth. The script will compare the date of birth with the current date and print “Happy Birthday” if appropriate
- The structure of the script is as follows:

```
Read the user's date of birth in the format YYYY-MM-DD
Get today's date in the same format
Strip the YYYY- component from both dates, leaving MM-DD
If the birthdate matches today's date
    Print “Happy birthday”
    Set a return status of 0 (true)
Else
    Set a return status of 1 (false)
Exit with the appropriate return status
```

A script to compare dates (continued)

- Things we need to know for the script:
- Bash provides a special form of variable substitution which will strip out the leading part of a string based on a regular expression match

`${VARIABLE#pattern}` If the pattern matches the beginning of the variable's value, delete the shortest part that matches and return the rest

```
$ A=cuckoo-clock
$ B=${A#*-}
$ echo $B
clock
```

Delete the leading part of the variable up to and including the first '-'

```
$ TODAY=$(date -I)
$ TODAY=${TODAY#*-}
$ echo $TODAY
04-07
```

`date -I` returns today's date in YYYY-MM-DD format

MM-DD only

A script to compare dates (continued)

Here is the completed script (`birthday.sh`)

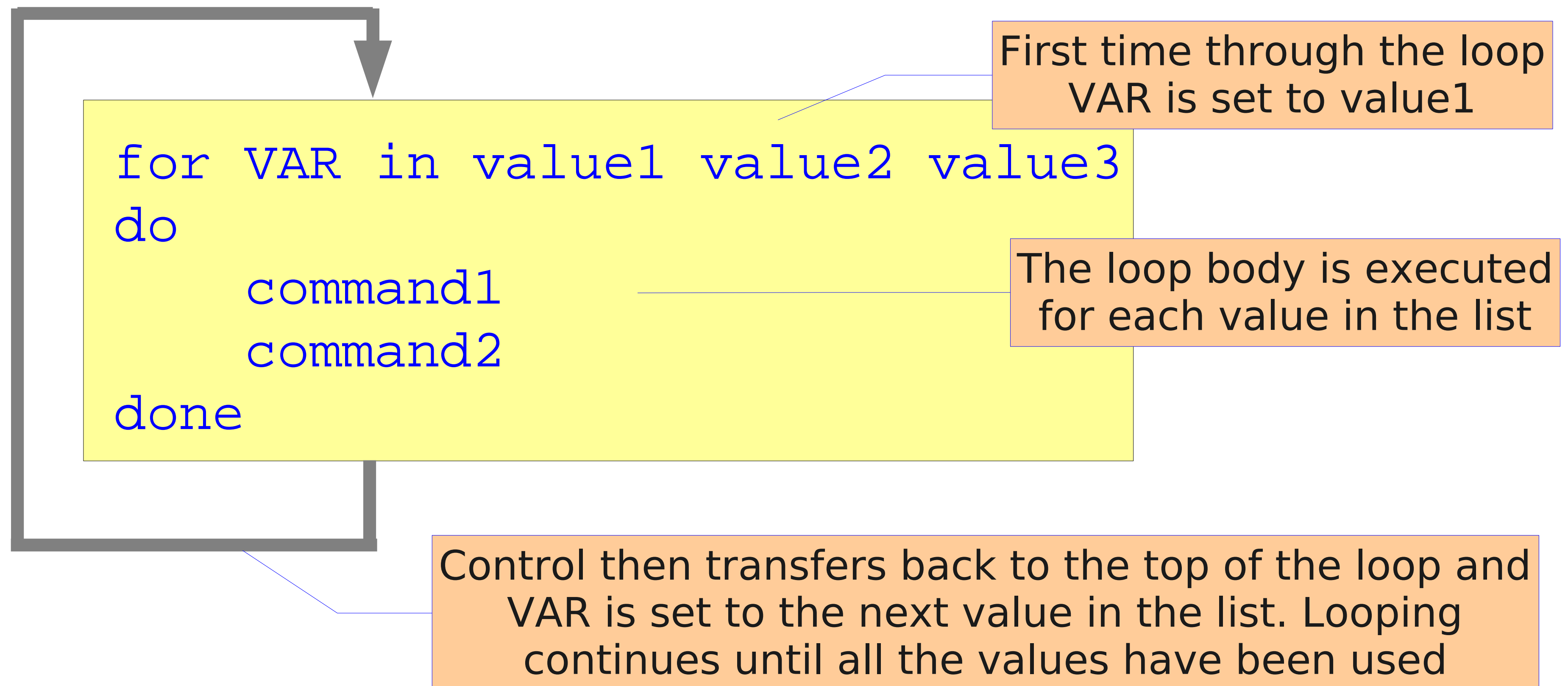
```
#!/bin/bash
echo -n "Enter your date of birth YYYY-MM-DD: "
read BIRTHDAY
BIRTHDAY=${BIRTHDAY#*-}
TODAY=$(date -I)
TODAY=${TODAY#*-}
if test $BIRTHDAY = $TODAY
then
    echo "Happy Birthday!"
    STATUS=0
else
    STATUS=1
fi
exit $STATUS
```

Exercise

- Write a script that takes a single file name argument from the command line and outputs a message to say if the file exists or not
  - Hint: use `if ... else`, and the `test` command
- **Bonus**: Extend the script so that it verifies that exactly one argument was passed, and prints an error message if not
  - Hint: the variable `$#` returns the number of arguments
  - The error message should, preferably, be redirected to `stderr`

Processing a list with a `for` loop

- A `for` loop allows a script to process a list of elements



Example of a `for` loop

- This example shows a `for` loop entered directly from the command line with the list of values specified explicitly
  - Note the appearance of the shell's secondary prompt, '>'

```
$ for i in 1 2 3 4 5 6 7 8
> do
>   ping -c1 earth$i
> done
```

- The loop can be written more compactly like this:

```
$ for i in 1 2 3 4 5 6 7 8; do ping -c1 earth$i; done
```

Generating the list for a `for` loop

- It's more common to generate the list for a `for` loop automatically

```
for host in $(cat host_list)
do
   ping -c1 $host
done
```

Loop over a set of host names held in a file

```
for file in *.old
do
   mv $file backup_directory
   echo archiving $file
done
```

Loop over a list of file names

```
for arg in $*
do
   echo processing argument $arg
done
```

Loop over all the command line arguments

Exercise

- Write a script to rename any files in the current directory that have upper case letters in their names, such that upper case letters are changed to lower case.
  - Hint: For safety, and to make testing easier, put an '`echo`' in front of the actual renaming command, so it is merely printed, not executed
  - Hint: The command `tr A-Z a-z` will change upper case letters in its stdin into lower case, and write the result to stdout. You'll need to think about how to invoke this within the context of the script.
- What would your script do if there were files called `FOO` and `foo`? If you have time, make the script safer by refusing to rename a file to a name that already exists.
  - Solution: `lowercase.sh`

Miscellaneous features

- Miscellaneous features
  - Aliases
  - Defining aliases
  - Startup files
  - Using shell scripts to automate system administration

Aliases

- An alias defines a shortcut name for a command
- The command `alias` (with no arguments) shows all your aliases:

```
$ alias
alias dir='ls -l'
alias ll='ls -l'
alias md='mkdir -p'
... plus lots more ...
```

- With one argument, `alias` shows you a single alias

```
$ alias md
alias md='mkdir -p'
```

- The command `type` will identify aliases and built-in commands

```
$ type -a ls exit
ls is aliased to `ls $LS_OPTIONS'
ls is /bin/ls
exit is a shell builtin
```

Defining aliases

- Aliases are defined like this ...

```
$ alias aliasname='command options'
```

- ... and undefined like this:

```
$ unalias ls
```

- Aliases are not inherited by subshells

```
$ alias ps="echo Hello"
$ ps
Hello
$ bash
$ ps
  PID TTY          TIME CMD
 1913 pts/2    00:00:00 bash
 2297 pts/2    00:00:00 bash
 2304 pts/2    00:00:00 ps
$ ^D
$
```

An alias is defined in the current shell

A subshell is started here

The alias is not inherited by the subshell

Exit back to parent shell

Exercise: Defining aliases

- The `cp` command recognises an option `-i` which causes it to prompt for confirmation before overwriting an existing file.

  Using the `alias` command, define an alias for `cp` such that it is always run with the `-i` option

- Test that the alias works, something like this:

```
$ cp /etc/motd junk
$ cp /etc/fstab junk
cp: overwrite 'junk'? Yes
```

- Start a new terminal window, and try the `cp` command again

  Is the alias defined in this new shell?

Bonus exercise: defining aliases

- Define an alias called `numproc` that reports the number of processes running on the machine, like this:

```
$ numproc
74
```

  - Hint: the command `ps ax` will list all running processes, one per line

- If you have more time: The `ps` command includes a heading line:

```
PID TTY      STAT   TIME COMMAND
```

  which will be included in the process count in a simple implementation of numproc

  Modify the alias to adjust the process count to ignore the header line

  - Hint: Look up the man page for `tail`, especially the '+' option

Startup files

- As we have seen, aliases and variables are local to the shell in which they are defined, they are not inherited by subshells
- Environment variables are inherited, but will not 'survive' if you log out and log back in
- To make aliases, environment variables, and other settings 'permanent', the commands that define them are placed into startup files that the shell reads automatically when it initialises
- Linux distinguishes between a *login shell* and a *non-login shell*
  - Login shells are those started directly by the login process
    - A login on a character based terminal
    - Or a login via a graphical display manager
  - A non-login shell is any shell started subsequently, for example:
    - If a new terminal window is started
    - If a shell script is executed

Startup files for login shells

- The following files are read when a login shell starts up
  - Not all of these scripts may be present; some exist for backward compatibility with other shells

/etc/profile	A system-wide configuration file read by all shells
/etc/bash.bashrc	A system-wide configuration file read only by bash; used to define aliases, set the command prompt, etc
~/.bash_profile	Per-user config file, not present by default in SuSE linux
~/.bash_login	Per-user config file, not present by default in SuSE linux
~/.profile	The main per-user configuration file used by SuSE linux; read by all shells

Startup files for non-login shells

- Non-login shells only read `~/.bashrc`
  - But remember, environment variables (such as your search path, `$PATH`) are inherited from the settings established in your login shell
- Sourcing of startup scripts
  - A change made to a file such as `~/.bashrc` will not be noticed automatically by the current shell
  - May need to make the current shell *source* the new file

```
$  source  .bashrc
```

  - The command '.' is a shorthand for `source`:

```
$  .  .bashrc
```

- For your convenience, in SuSE linux the default `~/.bashrc` file sources the file `~/.alias`, if it exists
  - This is a convenient place to put personal alias settings

Exercise

- Add your alias definition for `cp` (and the `numproc` alias, if you completed the bonus exercise) to the appropriate startup file so that it is 'permanent'
- Log out and log back in again. Verify that your aliases are available

Using shell scripts to automate system admin

- Professional linux administrators use shell scripts extensively to help automate system administration tasks, for example:
  - “Bulk” addition of user accounts
  - “Weeding” of redundant files from the file system
  - Installation and configuration of new software
  - Intrusion detection
- Command-line administration tools are more accessible to shell scripts than graphical tools
- Perl is an alternative scripting language favoured by some administrators for similar purposes
  - Especially effective for applications that involve processing, manipulation, or re-formatting of text files

Quiz

- What is the purpose of the line `#!/bin/bash` at the beginning of a script?
- Explain the shell variables `$#`, `$1` and `$*`
- Explain the sequence of events when this command is executed:

```
$ echo the date is $(date) | tr a-z A-Z
```

- On your machine, what would you expect this to print?

```
$ grep tux /etc/passwd
$ echo $?
```

- True or false?
  - Shell scripts are especially good for computationally intensive tasks
  - A shell script can be executed even if it does not have execute permission
  - Environment variables are inherited by subshells
  - Alias definitions are inherited by subshells