C Programming under Linux

P2T Course, Semester 1, 2004–5
Linux Lecture 4

Dr Graeme A Stewart

Room 615, Department of Physics and Astronomy

University of Glasgow

graeme@physics.gla.ac.uk

Summary

- Other Help Systems
- Shell Control Structures
- Shell Scripts

`http://www.physics.gla.ac.uk/p2t/`

`$Id: linux-lecture-04.tex,v 1.7 2004/10/26 11:34:21 graeme Exp $`

XEmacs Help

One of Emacs' claims was that it was 'self-documenting', and it includes a huge amount of on-line help.

The command `C-h` starts a help command – use `C-h ?` to see all the possibilities.

`C-h t` starts the XEmacs Tutorial – it covers, in more depth, a lot of the basic navigation and buffer commands described here.

`C-h i` starts the `info node` system, which is the heart of XEmacs help.

Info Nodes were a system of hypertext help developed by Richard Stallman before the advent of the web – they're incredibly useful, but a little weird to use.

SPC or PgDown	Move down 1 screen	DEL or PgUp	Move up 1 screen
n	Move to next node	p	Move to previous node
u	Move to parent node	l	Move to last node viewed
TAB	Move to next cross link	RET	Follow cross link

Info Nodes

When info starts it is in the ‘Directory Node’ – here all the info topics are cross linked. You can use the `s` command to search for a sting.

Of particular interest to us will be the `XEmacs` node; the `libc` node; the `Make` node and the `gdb` and `ddd` nodes. However, almost all GNU tools have info nodes.

Info nodes are particularly nice for learning a command – unlike man pages they contain tutorial information and examples.

To get access to info node help outside of XEmacs, use the standalone `info` command, e.g., `info gdb`.

N.B. KDE also has a help utility which provides a more ‘standard’ web browser interface to both man pages and info nodes. Activate it using the `Help` menu item:

`Help:` 

Shell Control Structures

The shell has a powerful set of *control structures*, which lift it from being a command line processor into a programing language in its own right.

- `for VAR in LIST; do SOMETHING; done;`
- `while CONDITION; do SOMETHING; done;`
- `if CONDITION; then SOMETHING;`
  `[else SOMETHINGELSE;] fi`
- `case VAR in; MATCH ) SOMETHING;; esac`

Here `SOMETHING` and `SOMETHINGELSE` can be multiple lines of shell code and commands – and multiple control structures can be embedded.

for VAR in...

The `for VAR in VALUE` structure creates a loop over a set of values:

```
$ for person in bob alice eve tom world; do echo hello $person; done
hello bob
hello alice
hello eve
hello tom
hello world
```

Note that the `person` variable doesn't take a `$` at the `for ... in`.

The list of values can come from a shell wildcard:

```
$ for file in *.data; do echo Data: $file; frobnicate $file | sort; done
```

(Even if `frobnicate` can take multiple arguments this is not at all the same as `frobnicate *.data | sort`!)

Processing Control Structures

The last example was getting rather hard to read on the command line, but the shell can read control structures in over multiple lines:

```
$ for file in *.data
> do echo Data: $file
> frobnicate $file | sort
> done
```

As with other long and continued lines the prompt changes.

Fortunately, if you typo, and the command does not run properly, then uparrow (↑) will recover the whole command, separated by semicolons.

Even so, it's clear that typing commands like this will quickly become tedious and prone to error. If we want to run a command like this time after time it's better to save it to a file, and run it from there.

Shell Scripts

When we have a set of commands which we want to run time and time again they can be assembled into a *shell script*, which can be run, just like any other command.

Shell scripts can be very simple:

```
$ cat simple-script-1
#! /bin/bash                          # <- Magic interpreter specifier
egrep -i knigits $@
```

But you probably want more comments than that:

```
$ cat simple-script-2
#! /bin/bash                          # <- Magic interpreter specifier
#
# Simple shell script example         # <- Comments!
# Prints any line containing
# "knigits" (case insensitive).
#
#  Graeme Stewart, 2003-10, v0.1
egrep -i knigits $@                   # <- $@ is a variable meaning
                                      #    "all my arguments"
```

However, they can also be. . .

A More Complicated Shell Script

. . . moderately complicated:

```
#! /bin/bash                          # <- Magic interpreter specifier
#
# Scripts can have comments in them
#
echo $0: You gave me $# arguments       # <- $# is "number of arguments"
for arg in "$@"; do                     #  \- $0 is the script name
  echo Processing argument $arg
  frobinicate $arg
  if [ $? != "0" ]; then              # <- $? is exit status of last command
    err=$?
    echo -n Disaster has struck the SS $arg:
    echo frobnicate said $err
  else
    echo Wonderous success reported from frobnicating $arg. Now we fankelise...
    fankelise $arg | grep "syrup" | sort | tail -1
  fi
done
echo Farewell dear invoker, I exit, with success
exit 0                                # <- Optional in shell scripts
```

Running a Shell Script

There are two ways of running a shell script:

- We can invoke a shell giving the script name as an argument: `bash path/to/myscript`
- We can execute the script directly if
  - it has the magic `#! /bin/bash` first line
  - it has been marked executable

```
$ myscript foo bar
myscript: You gave me 2 arguments
Processing argument foo
Wonderous success reported from frobnicating foo. Now we fankelise...
Ziggy says maple syrup is wonderful on ice cream!
Processing argument bar
Disaster has struck the SS bar: frobnicate said 5
Farewell dear invoker, I exit, with success
$ echo $?                          # <- $? works on the command line too!
0
```

Organising Scripts

Remember that for a script to be executed directly we have to either

- have the script in our PATH
- give the script's full path

It's a good idea to store your own scripts in `~/bin` and add this to PATH:

```
$ emacs ~/bin/myscript
$ chmod a+x ~/bin/myscript
$ echo $PATH
/usr/local/bin:/usr/bin:/usr/X11/bin:/bin
$ PATH=~/bin:$PATH
$ myscript
....
```

Usually PATH would be modified in `.bash_profile`.

if ... then ... else... fi

`if TEST; then COMMANDS; else OTHERCOMMANDS; fi`

This runs a series of commands conditional on `TEST`. If the *exit status* of `TEST` is `0`, i.e., success, then `COMMANDS` are run. If the optional `else` clause is present then `OTHERCOMMANDS` are run if `TEST`'s exit status is non-zero.

```
$ A=true
$ if [ $A = "true" ]; then echo "A is true"; fi
A is true
```

As in the above example, `TEST` is usually a shell builtin called, remarkably enough, `test`. `test` has two syntaxes:

- `test EXPRESSION`
- `[ EXPRESSION ]`

The second form, `[ EXPRESSION ]`, is much, much more common. (And the spaces around `EXPRESSION` are *required*.)

test

`test` can test a wide variety of things:

Filesystem Tests		String Tests	
`-f FILE`	Is `FILE` a regular file?	`S1 = S2`	Is `S1` equal to `S2`?
`-d FILE`	Is `FILE` a directory?	`S1 != S2`	Is `S1` not equal to `S2`?
`-r FILE`	Is `FILE` readable?	`-n S1`	Is `S1` non-zero length?
`-w FILE`	Is `FILE` writeable?	`-z S1`	Is `S1` zero length?

Combining Tests

`TEST1 -a TEST2`	Are `TEST1` and `TEST2` true?
`TEST1 -o TEST2`	Is `TEST1` or `TEST2` true?

See the `bash` man page or `help test` for a full list of tests.

Some TEST Examples

`[ -f datafile ]`	Is `datafile` a file?
`[ -d ~/bin ]`	Is there a `bin` directory in my home?
`[ -w .bash_profile ]`	Can I write to `.bash_profile`?
`[ $meal = "spam" ]`	Is `$meal` `"spam"`?
`[ "$A" != "$B" ]`	Is `$A` not equal to `$B`?
`[ -r $CF -a $CF != "$HOME/.noconf" ]`	`$CF` is readable and not `$HOME/.noconf`

The Tokeniser, Expansions and Quoting

Why does the [ ] need spaces around it? Why do variables with spaces need to be quoted? These apparent oddities can be understood by appreciating the order in which the shell's parser processes input and then performs substitutions upon it. See `info bash`, but essentially:

- The shell *tokenises*, reading each character and processing it and deciding if it's a word or an operator (e.g. >).
- The shell performs certain *substitutions* on words (e.g. variable substitution, like `$HOME` becomes `/home/graeme`.
- Then the shell performs *word expansion*, using the value of `$IFS` to split expanded *unquoted* words.
- As part of this process unquoted empty strings are discarded.
- Finally, the shell executes the commands or builtins or assigns values to variables as requested.

The Tokeniser, Expansions and Quoting (cont.)

Among the most common ‘gotchas’ in shell programing are:

- Not realising that expansions resulting in spaces will end up as separate *words* after word expansion, unless they are quoted.
- Forgetting that unquoted empty strings are ignored after word expansion.

For this reason it’s highly recommended to *quote all parameters passed to test*.

```
$ g=""
$ [ $g = "bob" ]     # ERROR - this expands to [ = "bob" ], a syntax error
bash: [: =: unary operator expected
$ [ "$g" = "bob" ]   # CORRECT - this expands to [ "" = "bob" ]
$ g=star bellied   sneetch
bash: bellied: command not found
$ g="star bellied   sneetch"
$ [ $g = "bob" ]     # ERROR - expands to [ star bellied   sneetch = "bob" ]
bash: [: too many arguments
$ [ "$g" = "bob" ]   # CORRECT - $g treated as a single ‘word’
$ echo $g - "$g"
star bellied sneetch - star bellied   sneetch  # Spot the difference!
```

while ... do ... done

`while CONDITION; do SOMETHING; done`

This shell control structure executes the code `SOMETHING` while the exit status of the `CONDITION` command(s) is 0 (i.e., successful – just like `if`).

```
#! /bin/bash
#
# while loop example:
#  iterate while $exit_flag is 0
#
exit_flag=0
while [ $exit_flag = "0" ]; do
  data_process $data_dir
  if [ $? != "0" ]; then
    exit_flag=1
  fi
done
```

Special Shell Variables

The shell has some special variables which are very useful for scripting:

`$?`	The exit status of the last foreground command
`$0`	The name of the shell script
`$1`	The first positional paramater (i.e., argument)
`$N`	The Nth positional parameter (i.e., argument)
`$@`	All the positional parameters
`"$@"`	All the positional parameters quoted (i.e., `"$1" "$2" "$3" ...`)

(As ever, the `bash` man page or info node documents the full list.)

shift

When processing arguments in a shell script the builtin `shift` can be very useful. It renames positional parameter `$2` to `$1`, `$3` to `$2`, etc. Positional parameter 1 is swallowed by the void.

```
#! /bin/bash
#
# Process until we run out of options
while [ -n "$1" ]; do
  foo $1
  shift
done
```

`shift N` moves the positional parameters by `N`, instead of 1.

case VAR in; esac

The last major shell flow control structure is `case VAR in; esac`. This is the shell's 'switch' statement. There then follow multiple pattern matches tested against VAR and series of commands to execute if the match is true.

```
case OPTION in
 -v )
    VERBOSE=1
    ;;
 -q | --quiet )
    QUIET=1
    ;;
 -* )
    echo Error: $OPTION is not an option I know
    exit 1
    ;;
 * )
    echo Error: Expected option string in $OPTION
    exit 2
    ;;
esac
```

case VAR in; esac (cont.)

Note that

- The list of commands to be executed are ended with a double semicolon.
- That multiple matches are provided by using the pipe, | (think or).
- That wildcards are acceptable in the pattern – so matching * is a ‘default’ action.

Only the first match’s commands are executed and it’s not an error to match nothing.

Nested Flow Control

All of the shell's flow control statments can be nested just as one would expect:

```
#! /bin/bash
while [ -n "$1" ]; do
  case $1 in
  -q | --quiet )
    verbose=0
    shift
    ;;
  -f | --file )
    file=$2
    if [ ! -r "$file" ]; then
      echo "File '$file' is not readable."
      exit 1
    else
      frobnicate "$file"
    fi
    shift 2
    ;;
  esac
done
```

Special Parameter Expansions

There are a lot of ways of modifying how parameters are ‘expanded’, but the most useful are:

- `${VARIABLE%STRIP}` If the string `STRIP` matches the end of `$VARIABLE` then expand to `$VARIABLE` with `STRIP` removed.
- `${VARIABLE#STRIP}` If the string `STRIP` matches the beginning of `$VARIABLE` then expand to `$VARIABLE` with `STRIP` removed.

In both cases if `STRIP` doesn’t match, just expand to `$VARIABLE`.

```
$ ls *.jpg
boris.jpg       malcom.jpg          zygote.jpg
$ for orig in *.jpg; do dest=${orig%.jpg}.png; echo convert $orig $dest; done
convert boris.jpg boris.png
convert malcom.jpg malcom.png
convert zygote.jpg zygote.png
```

Notice that the technique of `echo`ing a command instead of executing it is very useful when using more complicated paramater expansions.

Copyright