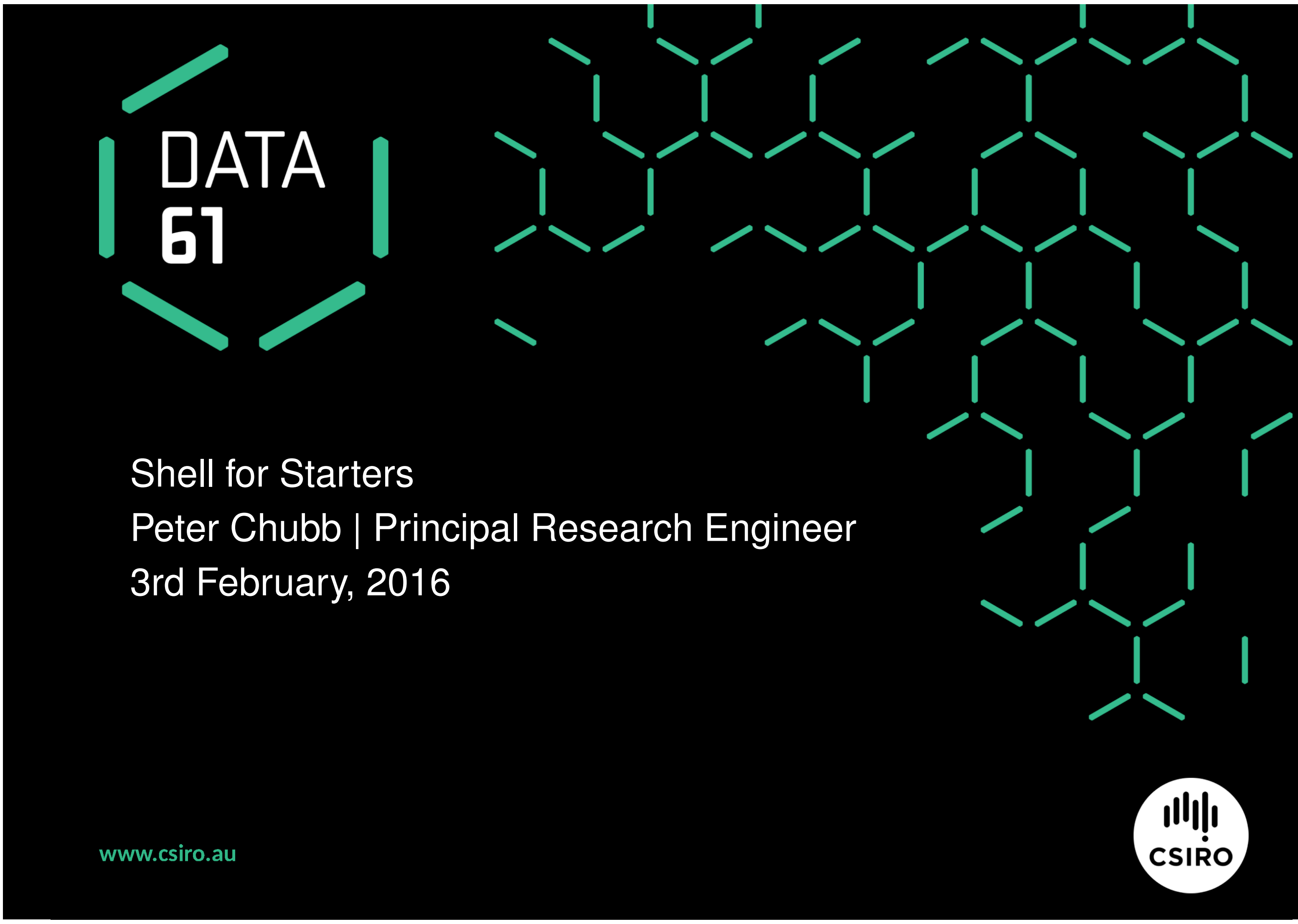


DATA 61

Shell for Starters

Peter Chubb | Principal Research Engineer

3rd February, 2016

www.csiro.au

CSIRO

This is a tutorial on the Bourne shell for people who want to start using the shell for programming as well as a user interface. It has been given in various venues; this particular edition is for Linux.conf.au 2016.

This is a tutorial

If you do the exercises you will remember

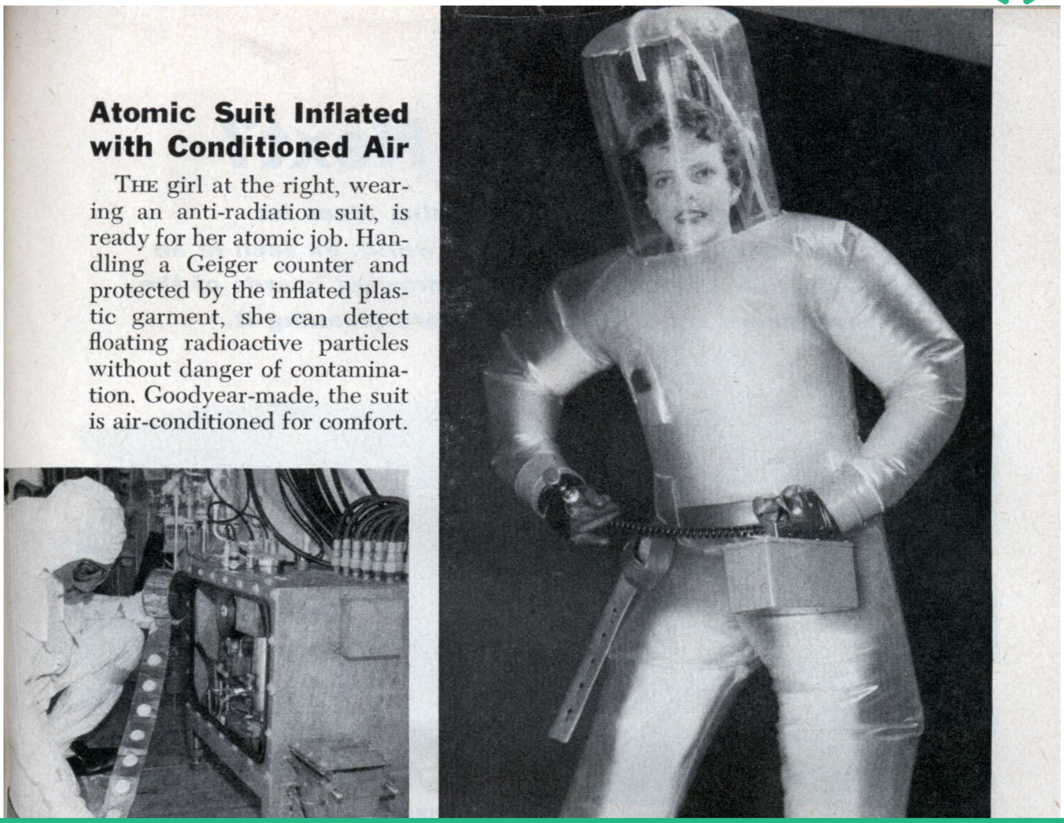
10 times more

than if you sit there twittering or reading your email.

What's the shell?

What’s the shell?

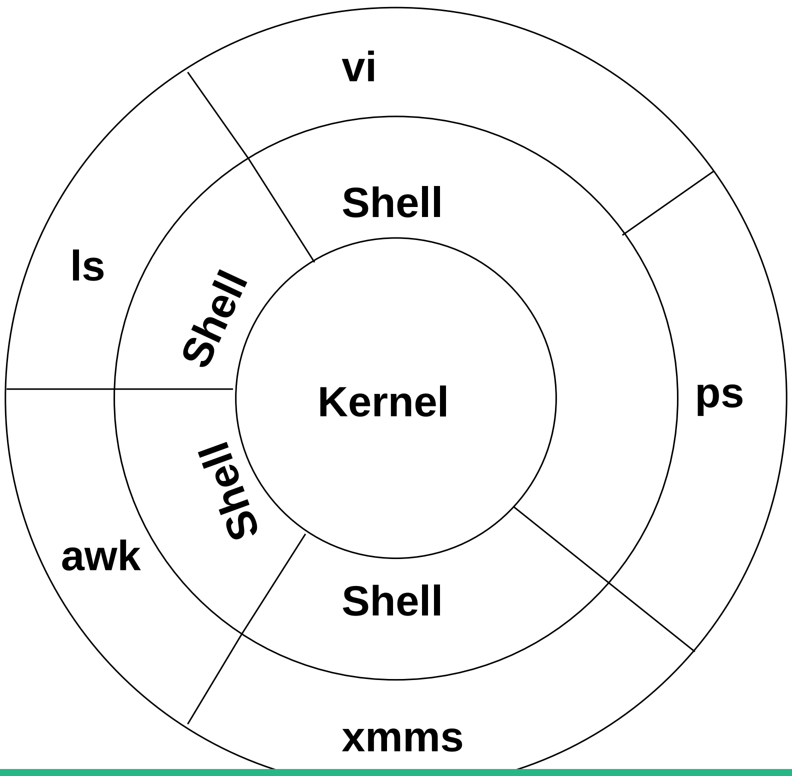
What’s the shell?



Atomic Suit Inflated with Conditioned Air

The girl at the right, wearing an anti-radiation suit, is ready for her atomic job. Handling a Geiger counter and protected by the inflated plastic garment, she can detect floating radioactive particles without danger of contamination. Goodyear-made, the suit is air-conditioned for comfort.

What’s the shell?



What’s the shell?

What's the shell?

There are *many* shells

- ➜ csh
- ➜ tcsh
- ➜ Bourne shell
- ➜ Korn shell
- ➜ Bourne-Again shell (bash)
- ➜ zsh
- ➜ etc., etc.

Usually means some kind of command interpreter.

A UNIX system has a *kernel* (the operating system proper) and around it, the *shell*. The shell is just a program that provides a user access to the kernel's functionality in a way that makes sense for humans. It can be *anything* — but is generally a general-purpose command-line interpreter and programming language. The shell is the program that's executed for you to interact with when you log into a computer running a UNIX-like operating system via a serial console or ssh etc.

The shell is usually dash or bash these days. There's also the C shell (**`csh`** or **`tcsh`**) that has a C-like programming interface. It's not often used now, as the POSIX-standard korn-shell has overtaken it in functionality and convenience; and **`bash`** is a free implementation that implements the 'history' part of the **`csh`** as well as a superset of the POSIX-standard.

Once upon a time, the shell was special — some things could be done only via the shell, or a similar privileged programme. This view of the shell persists in some other operating systems, such as VMS and OS-360 (where the shell is called, *JCL*). However, in a POSIX system the shell is unprivileged, as everything one needs to do to create files, hook them up to I/O descriptors in a programme, and start and stop processes can be done unprivileged.

What's the shell?

- Shell provides user interface
- Also a programming language
- Allows *toolkit* approach

We cover a subset of the POSIX 1003.2 Bourne/Korn shell.

Although the shell is primarily a user interface, it is also a multi-threaded programming language in its own right, with control-flow operations, definable functions, string parsing, exception handling, and I/O. Most of these features exist to allow the shell to

- control the *environment* within which programs run, and
- glue various other programs together

What’s the shell?

Every UNIX process has:

- an exit status
- a standard input (fd 0)
- a standard output (fd 1)
- a standard error (fd 2)

To facilitate being glued together, almost all UNIX programs behave in a similar way. They have an *exit status*, which is (conventionally) zero for success, and non-zero for an error.
For instance, **`grep`** exits zero if it finds anything, non-zero if it fails to find anything.
UNIX programs also generally act as *filters* — they read from standard input if they are not given any files to operate on, and write to standard output. Informational and error messages are separated out onto a separate file descriptor, so they don’t get mixed up with the ‘real’ data.

First shell program

In a file called '**`bang`**':
If you don't have **`sysvbanner`** installed, use **`echo`** instead.

```
sh ./bang
chmod +x ./bang
./bang
```

```
#!/bin/sh
X=9
tput clear
while test $X -gt 0
do
    tput cup 5 0
    tput ed
    banner "  $X"
    X=$(expr $X - 1)
    sleep 1
done
```

This program illustrates several things.
The first line, `#!/bin/sh` tells the kernel that this script is to be passed to the program `/bin/sh` for execution.
The next line sets the variable `X` to value 9.
The `test` command allows a great many different tests to be made. In this case we're testing whether the *value* of `X` is greater than zero. You can use square brackets `[...]` as an alias for **`test`**.
In the body of the `while` loop, four separate programs are invoked.
`tput clear` sends characters to the terminal that clear the screen.
`banner` prints its argument large. If you do not have the `sysvbanner` package installed, you could use `echo` instead: `echo` prints its arguments to standard output. `sleep` just sleeps for a number of seconds; in this case, one second.
The most complicated part is the `$()` construct. The stuff between the parenthesis must be a valid shell command — in this case, `expr`. The output from that command is then used as a word and parsed by the shell. The net effect in this case is to increment `X` by one.

Note that the words in red are all external commands; the rest is shell glue to join them into something ‘useful’.

New commands introduced (read about them in the on-line manual):
`man test` for example

test — test something.

tput — send control characters to the screen

sleep — delay for a number of seconds

expr — evaluate an expression

banner — print something large

echo — print something

First shell program

Grab
http://nicta.info/lca16

Unpack with:

```
tar xzf shell-tut-examples.tgz
cd shell-tut-examples
```

Finding out more ...

`man test`
to see what else you can test for...

`man -k` *keyword*
to search the manuals.

`man sh`
to see what the shell can do as glue...

How the shell reads commands

One line at a time

1. Tokenise
2. Expand Parameters and variables
3. Expand globs (wildcards)
4. command (backtick) substitution
5. Split into words using $IFS
6. Split into jobs
7. Execute commands

When the shell reads a line, it

1. Splits the line into *tokens*
2. Expands brace expressions[1]
3. Replaces variables with their values, and performs command substitution,
4. unquoted tokens are *globbed*
5. word splitting, this time using IFS, and stripping quotation marks.

A *brace expansion* is denoted by open-brace `{`, then characters, then the closing brace `}`. Between the braces are either a range, or a comma-separated list of alternatives. The word containing a brace expansion is repeated once for each alternative, or for each member in the range. Beware: brace expansion is not portable!

[1] Brace expansions are present in many shells, but not all. They are not mentioned by the POSIX specification

For example,

`{a..k}` is expanded to	**`a b c d e f g h i j k`**
`{a,b,c}`	**`a b c`**
`file{0,1}{0..2}`	**`file00 file01 file02 file10 file11 file12`**

Try it out, using **`echo`**.

How the shell reads commands

```
|ls|-l|/home/fred/aardvark|/home/fred/antelope|
|/home/fred/b*|/home/fred/colobus|
```

Then search `$PATH` for `ls`

```
 execl("/bin/ls", "ls", "-l", "/home/fred/aardvark",
"/home/fred/antelope", "/home/fred/b*",
"/home/fred/colobus", NULL)
```

Warning brace expansion is not present in all shells

After parsing the line, special characters are operated on, and finally the first word on the line is examined. If it is a *built-in command* or shell function name, the shell performs the operation directly; otherwise it searches the variable **`$PATH`** for an executable file named by the first word.

A Second Shell program

Put this into a file called ‘e’:

```
#!/bin/sh
for x
do
    echo "$x"
done
```

Then try (see if you can guess what the result will be beforehand):

```
$ chmod +x ./e
$ ./e a b "c d" 'e f' g\ h
$ X="A shell variable"
$ ./e $X "$X" '$X' \$X
```

By printing each argument, one at a time, on separate lines, you can easily see the effect of the quotation marks. Try also with the quotation marks around `$x` removed inside the `for` loop.

Standard File Descriptors

0 Standard Input
1 Standard Output
2 Standard Error

➜ Inherited from parent
➜ On login, all are set to *controlling tty*
➜ Other open files get other numbers

There are three file descriptors with conventional meanings. File descriptor 0 is the standard input file descriptor. Almost all command line utilities expect their input on file descriptor 0.
File descriptor 1 is the standard output. Almost all command line utilities output to file descriptor 1.
File descriptor 2 is the standard error output. Error messages are written to this descriptor so that they don't get mixed into the output stream. Almost all command line utilities, and many graphical utilities, write error messages to file descriptor 2.
Open file descriptors for a process are inherited from the process's parent.
When you first log in, or when you start an X terminal, all three are set to point to the *controlling terminal* for the login shell.

Redirection

environment command redirections

FOO=bah *command* 3>foo 2>&3

```
fd = open("foo", O_CREAT|O_TRUNC);
dup2(fd, 3); close(fd);
dup2(3, 2)
```

A shell command consists of environment settings (which usually must be first on the line), redirections (which are interpreted as they are encountered) and the command and its arguments. Commands can be combined into jobs using `||`, `&&`, and `|`; and can be grouped using parentheses `()` and braces `{}`.
Here, the environment variable **`foo`** is set to have value **`BAH`** for this command only.
And file descriptors 3 and 2 are set to output to file `"`**`foo`**`"`.

Redirection

General Form:

N>*file*	*file* created, fd *N* attached to it
N>`&`*n*	fd *N* duplicated to *n*
N>`&-`	fd *N* closed

(and the same with < for input files, and >> for append)

One common idiom:

command `2>&1 |` *command2*

to send stdout and stderr to command2's standard input.

N is assumed 1 for output, 0 for input.

The form *N*>*file* deletes file **`foo`**, opens it for output, and makes file descriptor *N* refer to it.
The form *N*>`&`*n* makes *N* a dup of `n`.
The form *N*>`&-` closes *N*.
If *N* is omitted in these three forms, file descriptor 1 is changed.
Using < instead of > allows manipulation of input file descriptors.
And you can use >> to open an output file for appending: each write operation will first seek to the end of the file.
The *pipe* (|) symbol says to start the commands on both sides of it at the same time, connecting the left hand side's standard output to the right hand side's standard input.
By adding `2>&1` in there, *command*'s standard error is also sent through the pipeline.

Redirection

Quick and Dirty:

To create a file:

```
$ cat > /tmp/foo
some random text
^D
$
```

What does

```
> /tmp/foo
```

do?

What does

```
exec > /tmp/foo
```

do?

If you want to create a file fast and are a reasonably accurate typist, you can just use `cat` and redirect the output to a file.
The shell performs redirections *before* it attempts to run any command; if there is no command, then the effect is to open and close the file, possibly wiping out its contents.
To change the shell's standard I/O, use, e.g., `exec 2>/tmp/log`

xargs: converting input to arguments

```
$ xargs sh ./e
```

type some lines of input ending with control-D at the start of a line

```
$ xargs -d '\n' sh ./e
...
$ xargs echo
...
```

The **`xargs`** command splits its input into words, and then invokes its first non-option arguments as a command with those words passed as arguments.
By default it splits on whitespace; you can set the delimiter with the **`-d`** option, or sepcify **`-0`** to use null-delimited words.
You can specify how many arguments are given; the most common are either to allow **`xargs`** to use the maximum argument buffer size for the system you are on (the default), or specify **`-n 1`** to give exactly one argument.

Jobs, processes etc

- Jobs separated by newlines, semicolons ‘`;`’ or ampersands ‘`&`’
- semicolon or newline for sequential operation
- ampersand for parallel operation

```
$ long_running_process & another process &
```

starts both programs in parallel.

Each separate program as it is run becomes a process. Processes are grouped by the shell; typically each process in a pipeline belongs to the same process group. Process groups can be manipulated as a bunch.
Some shells do job control. Each process group run under a job control shell becomes a job; it can be started, stopped and moved from foreground to background, etc., under control of the shell.
Jobs are separated by newlines, semicolons or ampersands.
So for example:

```
$ make world > outfile 2>&1
```

starts a `make` process with standard output and standard error redirected into `outfile`, and does not wait for it to finish.

Jobs, processes etc

- ➜ Each process has an *exit value*
- ➜ Zero means success, non-zero failure.
- ➜ Special variable `$?` contains last exit value
- ➜ Can use `&&` and `||` to join commands

Every process has an exit value, which is an eight-bit value. In a C program it is the low eight bits returned from **`main()`**, or the argument to **`exit()`**.
In a shell script, you can do **`exit 0`** or **`exit 1`**; if you don't explicitly invoke **`exit`** the exit value of the shell is the exit value of the last command.
The exit status of a pipeline or job is the exit status of the last command run.
Two programs usually available are `/bin/true` (which always exits zero) and `/bin/false` (which always exits non-zero).

Jobs, processes etc

```
$ true || false
$ echo $?
0
$ true && false
$ echo $?
1
$ false | true
$ echo $?
0
```

You can play with `true` and `false` to see how `||` and `&&` work. The exit status of a pipeline is the exit status of the rightmost command.

Jobs, processes etc

```
grep peterc /etc/passwd > /dev/null 2>&1 ||
    echo >&2 "peterc not in password file"
```

In this example, we invoke **`grep`** to search for the string **`peterc`** in the password file. We redirect all the output to **`/dev/null`**, so we never see it. **`grep`** exits non-zero if it does not find the string; **`||`** says ‘do the next command *only* if the previous command exited non-zero’. So the effect is to put a message onto standard error if the string **`peterc`** is not in the password file.

Jobs, processes etc

PATH:

```
PATH=/bin:/usr/bin

CDPATH=.:$HOME

MAILPATH=/var/mail/$USER:$HOME/Mail/inbox
```

In general, a **`PATH`** variable consists of a list of directory names separated by colons. **`PATH`** itself contains the list of directories to search for executable files; but there are many other **`PATH`**-like variables that are interpreted by various programs. For example, **`CDPATH`** gives a list of directories to search when changing directory in the shell. Be careful using this; if the current directory (`.`) is not in **`CDPATH`** many things can break. By the same token, `.` should *not* usually be in **`PATH`** — especially if you are running as the superuser. It is too easy for someone to put a spoof version of a commonly user program (e.g., **`ls`**) in some directory then get you to visit it.
`MAILPATH` is obeyed by some shells (including *bash*). It contains file names, not directory names. When any of the filenames in **`MAILPATH`** change, the shell prints a message. While it is designed to be used for mail notification, it can actually be used to notify changes on any file.

Exploring PATH

Put this into a file called ‘wh’:

```
#!/bin/sh
IFS=:
for arg
do
   for dir in $PATH
   do
      test -x $dir/$arg && echo $dir/$arg
   done
done
```

Explore by the usual `chmod +x wh; ./wh which`
This program not only illustrates `PATH`, it also shows off the Internal Field Separator variable. After the shell has expanded any variables, it splits the line into words based on the value of `IFS`. By default, IFS contains whitespace: space, tab and newline. If unset, this is what the shell will use.
In this program, we set IFS to a colon, so the PATH variable after expanding will be split into one word for each component. Remember, the shell has tokenised the program before this stage, so only the `PATH` variable is affected.

set

set [*-option*] [--] *arg1 arg2* ...

	display all variables and their expansions
–x	display commands before executing
–v	display commands as read
–e	exit immediately if any command exits non-zero

`set` allows you to provide options and arguments to the shell as if to the command line. For example, `set -x` inside a script is the same as invoking the script with `sh -x`.
Without arguments, **set** displays all current arguments and their expansions.

`set`

E.g., to extract 7th field from `/etc/passwd`:

```
#!/bin/sh
x=`getent passwd "$1"`
IFS=':'
set -- $x
echo "$1 has shell $7"
```

` `` ` is the same as `$()`

This snippet introduces backticks, which are an alternative for the `$()` notation seen earlier. `getent` looks up a key in a UNIX database — in this case, the password database, which could be a file in `/etc`, or something distributed by LDAP or NIS. `passwd` has a standard format: individual fields are separated by colons. The first field is the login ID, the seventh is the user's shell. Thus, `set` here sets the arguments to the shell script to the seven components of the password entry.
Take a look in `shell-tut-examples/Finger` for some more use of this construct.

Variable expansion

- Variables expanded by $.
- Can delimit variable name with braces, `${`*variablename*`}`
- Special Forms

`$#`	number of arguments to shell	`$!`	PID of last asynch job
`$0`	name of script	`$?`	exit status of last job

See the man page or Posix.2 for the full list of special forms

Variable names are any sequence of alphanumeric characters and underscore, starting with a alphabetic character or underscore.
If it isn't obvious where a variable name ends, use braces `{}` to delimit the variable name. For example,

```
dir=/bin/
echo ${dir}ls
```

Shell variable examples

```
$ foo=/home/fred
$ echo $foo
/home/fred
$ cd $foo
$ pwd
/home/fred
$ b=/home/fred/bin/prog
$ $b
... output of prog
$
```

You can assign any string to a shell variable. Some shell variables (e.g., **`PATH`**) have special meanings to the shell; others can be used as shorthands for anything you like. The shell replaces a variable name with its value if and only if the variable name is prepended with a dollar sign.
Shell variables are local to the shell they are set in. They have three states:

1. unset
2. null
3. set with a value.

In addition, shell variables can be *exported* to child processes. The **`export`** operator marks a variable as part of the environment.

Shell variable examples

The Environment:

```
$ foo=bah
$ set | grep foo
foo=bah
$ env | grep foo
$ export foo
$ env | grep foo
foo=bah
```

`set` is a shell builtin that, when used without arguments, displays all currently set variables. It can also be used to set positional parameters or shell options. **`grep`** (the name is taken from the **`ed`** command **`g/r.e./p`**) searches its input for the occurrences of a regular expression (RE) and prints any lines containing the RE.
The result is that

```
        set | grep foo
```

will print the name/value pair for any shell variable with **`foo`** in its name or value.
`env` is a program for printing and manipulating a process's environment. The *environment* is a collection of name/value pairs that is available to a process, and to all of its children (unless special steps are taken).

Special parameter forms

${parameter:-word} expands to `$parameter` if parameter is set and non-null; otherwise to `word`.

${parameter:=word} expands to `$parameter` if parameter is set and non-null; otherwise `parameter` is set to the expansion of `word`, and that is then used as the result.

Omitting the colon means test only for set, and allow NULL values.

When you're using variables there are a few special things you can do. They're most useful in scripts, especially together with the **`:`** (do nothing) and **`eval`** (reread, and reparse a line) operators.
There are a number of other special forms; see the man page for details.

Special parameter forms

Setting defaults:

```
: ${CC:=gcc-4.7}
```

Set CC to the string `gcc-4.7` only if it is not already set.

`:` is the no-op shell built-in.

It's often useful to set up overrideable default values in a script.
The Null command `:` does nothing, but any side effects in expanding its arguments still take place. `:` always exits zero.

Quoting

Three forms:

1. Backslash quotes next character. `a=\$`
2. Single quotes `'  '` prevent all expansion; and everything between the quotes is a single word.
3. Double quotes `"  "` allow parameter and command substitution, and everything between the quotes is a single word.

Usually the shell interprets whitespace and various other special characters. If you want them to be used as arguments to your command you have to *quote* them to protect them from the shell.
Any single character can be quoted with a backslash `\`. To enter a literal backslash it too needs to be quoted: `\\`.
Single quotation marks prevent interpretation of anything inside them. To include a quotation mark in a word, do `'Alan O'\''Brian'`. The single quotes prevent the space from separating the words; as there's no space between `O` and `\'` the single quote is included in the word along with `Brian`.
Double quotation marks `"` prevent whitespace and some other special characters such as single quote from being interpreted, but dollar and backtick expansions still happen.

Quoting example

```
$ h=" foo bah"
$ echo "$h"
   foo bah
$ echo \$h
$h
$ echo $h
foo bah
$ echo '$h'  Try these yourself
$ echo X$h
$ echo $hX
$ echo ${h}X
```

Try the examples. For each one, work out what you think the result will be before you try it.

Grouping

Parentheses:

- ➜ Parentheses start a subshell
  - ➜ variable changes not visible to parent
  - ➜ current working directory can be different from parent
  - ➜ Can redirect output as a group
  - ➜ Example:

```
(cd /; echo *; ls; ) | less
```

Parentheses start a subshell: in logical terms the shell forks a copy of itself to run the commands inside the parentheses. (Some shells do fork themselves for this; others emulate the behaviour to avoid having to create another process).
You can put a subshell into the background using `&`, and manipulate it exactly as for any other job.

Grouping

Braces:

➜ braces start a simple group

  ➜ Variable changes *are* visible

  ➜ CWD changes affect current shell

  ➜ Can redirect output from group

  ➜ Example:

```
 { echo \
"login:password:uid:gid:GECOS:name:home:shell";
cat /etc/passwd; } | less
```

Note here that the backslash removes any special interpretation of the new-line; without it, **`echo`** and the line starting **`"login`** would be treated as separate commands. Unlike parentheses, braces are not implemented with an implied **`fork()`**.

here documents

```
 >&2 cat «-AnythingGoes
This is an Error Message
that goes on for ages
and ages
AnythingGoes
```

A ‘here document’ is a way of providing input to a command from within a shell script (it works from the terminal too, but is not very useful there).
This example copies each line inside the here document onto standard error.

here documents

«*string*

stdin becomes everything from here to next occurrence of *string*

Can add a dash to ignore tabs at start of line:

«−*string*

Quoting any part of *string* prevents expansion in the here document

«\string

The shell performs its usual scan and replace operations unless *string* is quoted in any way. This allows shell variables and backtick commands to be used to customise the here document contents.

Example Pipeline

Build a wordlist:

```
cat /usr/share/doc/*/*.{txt,htm}|
tr '[:upper:]'  '[:lower:]'  |
tr -cs '[:lower:]'  '\n'  |
sort -u > /tmp/words
```

`tr` is useful little text transformation utility. It reads only from standard input and writes the transformed text exclusively to standard output. It takes two arguments: the 'from' and the 'to' strings. These can either be explicit sequences of characters (e.g., `tr cdefg 12345` to translate notes on a piano to fingerings starting at 'c'), ranges as in the example, or, in recent versions of tr, character classes. The flags 'c' and 's' complement the list, and 'squash' adjacent repeating changed characters to a single replacement.
The net effect is that the first invocation of 'tr' changes upper to lower case; the second converts any sequence of anything not a lower case alphabetical character to a single newline '\n'.
Then the result is sorted and duplicates thrown away ('-u' option).

shell control flow

- ➜ Shell has `while, if, until, case` and `for`.
- ➜ Control conditions are generally process exit codes.
- ➜ Most UNIX processes exit 0 for success.

The shell has a complete set of conditional constructs, including **`select`** which allows interactive input. Though unfortunately **`select`** is a bash-ism and may not be present in all shells.

shell control flow

```
if grep '^zoom$' /tmp/words >/dev/null 2>&1
then
    echo "Zoooooooommmmmm"
else
    echo "Slooooow"
fi
```

Here's a very simple example. If any line containing only `zoom` is in `/tmp/words`, then the shell will echo `Zoooooooommmmmm`.

shell control flow

for:

➜ **`for`** iterates over its arguments

```
for x in a b c
do
  echo $x
done
```

If there is no `in ...`, `for` iterates over `"$@"`.

The **`for`** command sets its first argument to each of its arguments after **`in`** in turn. If **`in ...`** is omitted, it iterates over the arguments to the shell. This command can also be used on a single line:

```
for x in a b c; do echo $x; done
```

Note the lack of a ';' between the do and the echo, if you add a semi-colon there it will cause an obscure error.

case and Option arguments

```
while getopts 'n:' c
do
   case $c in
   n)
      number="$OPTARG"
      ;;
   *)
      Usage
      ;;
   esac
done
shift $OPTIND
```

`getopts` is a shell built-in that allows easy option handling. Each time it is invoked, it steps through the values in `"$@"`. If the current argument starts with a dash, it's considered an option, and is split into individual options.
The two arguments to `getopts` are first a string, containing allowed options, and then a variable name. If an option takes an argument, it has a colon (`:`) appended, and if encountered, the optional argument is put into a shell variable called `OPTARG`.
Options can be stacked (so for example `-a -v` could be written `-av`). When the first non-option argument is encountered, `getopts` has a non-zero exit status, and sets `OPTIND` to the number of arguments handled. These can then be thrown away with `shift` before handling non-option arguments.
case does pattern matching. Each alternative is one or more shell glob expression, separated by | symbols.

Shell arguments

- ➜ Shell arguments available as `$*` and `"$@"` and as `"$1"` to `"$9"`
- ➜ Watch quoting!
- ➜ `shift` throws arguments away.

The first nine arguments to the shell (the *positional parameters*) are available as $1 through $9. These are just shell variables; the same quoting rules you've already learnt apply. To preserve whitespace and special characters, quote with double quotes.
The **`shift`** command throws away parameter 1 and relabels the others (so 2 becomes 1, 3 becomes 2, etc., and the previously unnamed tenth argument becomes $9).
Alternatively you can get at all the arguments at once with either $* (which does not preserve white space) or "$@" (which expands to "$1", "$2", "$3", . . .). "$@" is almost always the correct term to use, it works when there may be spaces in the arguments being processed.

Shell arguments

This is a variant on the echoline program we wrote earlier.

```
#!/bin/sh
while test "$1"
do
    echo "$1"
    shift
done
```

This example merely prints its arguments one per line. To try it out, copy it into a file, then change the mode of the file to make it executable, then run it. It does the same as the 'e' example earlier.

```
$ cat > el
#!/bin/sh
while test "$1"
do
   echo "$1"
   shift
done

control-D
```

```
$ chmod u+x ./el
$ ./ul 1 2 '3  4'
1
2
3  4
$
```

Example: `isup`

```
isup() {
  ssh "$1" sleep 1 &
  PID=$!
  (
     sleep 15 && kill -9 $PID 2>/dev/null
  ) &
  wait $PID
}
```

The `isup` script illustrates several features. It starts with the usual boilerplate (not shown in the slide), then has a function `isup()` that does most of the work of determining whether a host is up and listening on the ssh port.
It first starts `ssh` to the host to do a one-second sleep in the background. The process ID of the ssh process is saved in the shell variable `PID`.
Then a subshell is set up (note the parentheses), again in the background, to kill the `ssh` if it takes more than (in this case) 15 seconds. The process id of the subshell is stored in `PID1`.
The standard error of `kill` is redirected to `/dev/null` to make sure any inconvenient messages about `Process not found` don't confuse the human using `isup`.
The shell then `wait`s for the original process to complete.
If it has successfully connected and slept for a second, its exit status is 0; if it failed, or if it was killed after the 15 second timeout, its exit status will be non-zero. In either case, the exit status of `wait` will be the exit status of `ssh`. The 'exit' status of a shell function is the exit status of its last command, so the exit status of `isup()` is the exit status ot `wait`. Alternatively, the `return`

op allows an explicit status to be returned.
In the version of `isup` in the solutions tarball, that exit status is saved in `RET`, then the sleep and kill process is itself killed. Strictly speaking there's no need to do this, because after 15 seconds it'll die anyway, and if the `ssh` has finished, it'll do nothing.
Finally the function returns itself with the same exit status as `ssh`.

Example: `isup`

```
for x
do
  (
    isup "$x" && echo "$x" >&3
  ) 3>&1  </dev/tty >/dev/tty &
  PIDS="$PIDS $!"
done
```

The main part of the `isup` script comes next. It's a for loop that iterates over the arguments to the script, each of which should be the name of a host to test.
The `isup()` function is invoked in a backgrounded subshell, with standard input and output redirected to the terminal, allowing `ssh` to ask for passwords if necessary. The old standard output is duplicated onto file descriptor 3, so that the `echo` inside the subshell still writes to the original standard output.
Recall that the exit status of a job is the exit status of the last process executed in the job — in this case, if `isup` succeeds, then `echo` (which will always succeed).
The process ID of each background call to `isup()` is saved in the `PIDS` variable.

Example: `isup`

```
ret=0
for pid in $PIDS
do
        wait $pid || ret=1
done
exit $ret
```

Finally, we want the exit status of the whole script to be non-zero if any of the tested hosts is down. Unfortunately, `wait` on its own will always exit 0 if more than one process is waited for — which means waiting for each process one at a time.

Hints for Shell Programming

- `cat file | pipeline`
  Don't do it! Use
  `< file pipeline`
- minimise external programs ... try to do expensive operations **once only**
- If you're using `while read` you're probably doing it wrong.
- Rework problems as filtering problems where possible.

Lots of people use **`cat`** when they don't need to. Particularly in an embedded system where processes are expensive, try to use as few as processes as possible.
Also, try to rework problems as filtering problems, so that each piece of data gets handled only once.
Handling data with a shell `while` loop is usually extremely inefficient — it you're doing it for a procedure that loops more than a few times, you're probably better off using a different tool, or refactoring your program as a pipeline.

Filtering example

Counting number of shell scripts on system

```
find / -type f -perm +0111 -print0  2>/dev/null |
   xargs -0 file -N |
   grep 'text executable' | wc -l
```

Here's an example, to find all the shell scripts on a system. Look up **`find`** in the appendices — basically it starts at the root directory and for each regular file (**`-type f`**) that is executable (**`-perm +0111`**) print its name followed by a NUL('\0').
`xargs` reads NUL separated words, and piles them up as arguments to its argument. The effect is to call **`file -N`** on the files found by **`find`**.
`file` works out (using heuristics) what the files named by its arguments actually are. It prints a short description to standard out. The **`-N`** ensures that each result is on a single line.
`grep` then picks out all the files with **`text executable`** in their type.
Then **`wc`** counts words, lines and characters in its input ... in this case the **`-l`** option says count only lines.
The result is a count of the shell scripts on the system.

Filtering example

Finding missing shared libraries in a filesystem image

```
find . -type f -perm +0111 -print0 |
xargs -0 file -N |
sed -n 's/^\([^:]*\): ELF.*executable.*dynamic.*/\1/p' |
  xargs chroot . ldd 2>/dev/null | awk '
      NF == 1 {  filename = $1 ; next}
     /not found/ { print filename ": " $1; }'
```

This is a more complicated example, taken from a script to generate a root filesystem for an embedded system.
It's meant to be executed, as root, at the top of the new directory tree that will become the root filesystem.
`find` and **`xargs file`** are as before.
The next part is a **`sed`** command. It has the **`-n`** option so will print only lines that get changed. To be changed, a line has to match the regular expression — i.e., to be an ELF executable (i.e., a standard binary) that is dynamically linked.
The output of **`sed`** is a list of filenames.
The next **`xargs`** runs **`ldd`** on each filename in turn. **`ldd`** prints the names of all the shared libraries needed by a command.
The **`awk`** script (and I suggest you obtain the O'Reilly book on AWK; it's a very useful tool, but out-of-scope for this tutorial) collects the file name, then if for that filename a line is found that says 'not found' it prints the filename and the library that wasn't found.

Cleaning up your Droppings

```
tmpfile()
{
   T=$1$$
   TMPFILES="$T $TMPFILES"
}
trap 'rm -f $TMPFILES' 0

tmpfile foo # tmpfile doesn't work in a subshell
FOO="$T"
```

Works only if temp filenames do not contain special characters.

Often you want a temporary file in a shell script, where you can't use a pipe, and there may be too much data for a backtick expression. It's considered at best bad manners to leave droppings around in `/tmp` (or, worse yet, in the current directory). But you need to be able to cope even if the script is interrupted.
Enter `trap`. `trap` lets you run a command when a signal arrives, or when the script exits for whatever reason (of course, you can't catch signal 9, but you're not expected to clean up for a signal 9.)
The syntax is
`trap` *command signalnumber...*
When any of the signal numbers arrive, *command* is executed.
Note that in this case, signal 0 means, 'whenever the shell exits for whatever cause'.
Note the quoting: the single quotes mean that the shell does not expand the variable when trap statement is encountered, but when the trap actually happens.

A complete example: Finger

Look in the shell-tut-examples directory.

```
#!/bin/sh
progname=`basename $0`

error()
{
echo >&2 "$@"
exit 1
}

Usage()
{
error "Usage: $progname name"
}

fingerOne()
{
    key="$1"
    OIFS="$IFS"
    IFS=':'
    {
        getent passwd "$key"  2>/dev/null ||  error "Can't fin
    }  |  while read loginname passwd uid gid gecos home shell
    do
IFS=','
set -- $gecos
echo "Login: $loginname Name: $1"
echo "Directory: $home Shell: $shell"
echo "Office: ${2:-unknown}, phone (W)${3:-unknown}  (H)${4:-u
IFS="$OIFS"
export LC_TIME=C
ON="`who | grep $loginname`"
if [ "$ON" ]
then
    set -- $ON







    echo "On since $5 $4 $3 on $2 from $6"
else
    ON="`last -1 $loginname | grep -v wtmp`"
    set -- $ON
    if [ "$1" ]
    then
echo "Last on at $8 $4 $5 $6 from $3"
    else
echo "Never logged in"
    fi
fi
if [ -f $home/.plan ]
then
    if [ -r $home/.plan ]
    then
echo Plan:
cat $home/.plan
```

```
    else
echo "Private plan"
    fi
else
    echo "No plan."
fi
    done
}


if [ $# -lt 1 ]
then
Usage
fi

for login
do









    fingerOne  $login
done
```

This is a shell script that works like ‘finger’. It first saves the program name for error printing (`basename` strips off any directory name component from its argument; `$0` is the full name of the shell script).
It then sets up three shell functions. The first echoes its arguments to standard error, and exits with an error status.
The second calls the first with a usage message, using the saved program name as part of the message.
The third does most of the work. It takes an argument (the login name of the person being fingered), saves the `IFS` (recoolect, this variable contains the field separators), then calls `getent` to look the login name up in the password database (`/etc/passwd`, LDAP or NIS). If `getent` fails, it'll invoke the `error()` function to tell the user; otherwise, `getent`'s output is passed to a `while` loop.
`read` is a shell builtin. It reads a line from standard input, splits it according

to `IFS`, then assigns the first word to the variable named by its first argument, the second to the second, etc., and anything left on the line to the last argument. If the input line is empty, it exits with an error. The result in this case with the `while` is to iterate over all the values output by `getent`.
The `gecos` field of the password file contains some extra data, in comma-separated fields: the person's full name, office, and phone numbers. It's called GECOS because this info was used in the early days of UNIX to log into the Honeywell computer that was used fof printing, etc. That computer ran an operating system called GECOS (General-Electric's Comprehensive Operating Supervisor).
The script splits those with `set` — the double dash `-` serves to terminate any option arguments, so the remaining words in the gecos field are assigned to $1, $2, etc.
The same trick is used for parsing the output of `who` and `last`, except that the output of these commands is dependent on the time format. To avoid issues, the script sets the `LC_TIME` locale to `C`.
Finally, all the shell functions are done. The script checks to see if it has been

invoked correctly (does it have at least one argument?) then iterates over all the arguments and invokes the `fingerOne()` function.

Fables server

Create a script to print a single fable chosen at random from `aesop11.txt` in the examples directory.
Do:

```
telnet lemon.ertos.nicta.com.au 8500
```

to get an idea.

Fables server

- Could count the lines in the file, search backwards for start of fable, then print to end of fable. Too slow!
- Or build an index, then use the index to grab a fable

Fables server

Run `sh ./get-solutions`

See `Fables/findex` and `Fables/findex1`
Compare their speeds.

Take a look at `fable` and then enhance it to take an optional argument to specify a particular fable.

Useful tools for Shell

man read or search for manual entry

echo put arguments onto standard output

xargs convert stdin to arguments

test attributes of strings or files

sed Stream editor

expr Do arithmetic or string manipulation.

basename/dirname extract parts of file names

There are a number of very useful programs that should be in your toolbox. It is always worth learning these commands as they are on almost every single unix in existence including most embedded systems.