COMP(2041|9044) 24T2 — Shell

https://www.cse.unsw.edu.au/~cs2041/24T2/

Shells

- Shells are command interpreters
  - they allow interactive users to execute the commands.
  - typically a command causes another program to be run
  - shells may have a graphical (point-and-click) interface
    - much easier for naive users
    - much less powerful & not covered in this course
- command-line shells are programmable, powerful tools for expert users
- **bash** is the most popular used shell for unix-like systems
  - other significant unix-like shells include : **dash**, **ash** , **zsh**, **fish**
- we will cover the core features provided by most shells
  - essentially the POSIX standard shell features
- we use **dash** for scripts in this course
  - **dash** implements essentially the POSIX standard shell features
  - **bash** & **zsh** implement superset of POSIX shell features
  - **ash**, part of **busybox**, implements more-or-less the POSIX standard shell features
  - so scripts written for **dash** usually compatible with with **bash** & **zsh**, **ash**

What Shells Do

- Unix shells have the same basic mode of operation:

```
loop
   if (interactive) print a prompt
   read a line of user input
   apply transformations to line
   split line into words using whitespace
   use first word in line as command name
   execute command, passing other words as arguments
end loop
```

- shells can also be run with commands in a file
- shells are programming languages
- shells have design decisions to suit interactive use
  - e.g. variables don't have to be initialized or declared
  - these decisions not ideal for programming in Shell
  - in other words there have to be design compromises

Processing a Shell Input Line

- a series of **transformations** are applied to Shell input lines
  1. tilde expansion, e.g. **~z1234567** → **/home/z1234567**
  2. parameter and variable expansion, e.g. **$HOME** → **/home/z1234567**
  3. arithmetic expansion, e.g. **$((6 * 7))** → **42**
  4. command substitution, e.g. **$(whoami)** → **z1234567**
  5. word splitting - line is broken up on white-space
  6. filename expansion (globbing), e.g. ***.c** → **main.c i.c**
  7. I/O redirection e.g. **<i.txt** → stdin replaced with stream from **i.txt**
  8. first word used as program name, other words passed as arguments
- order of these transformation is important!
- not understanding order is a common source of bugs & security holes
  - shell is better-avoided if security is significant concern
- directories in **PATH** searched for program name

echo: print arguments to stdout

- **echo** prints its arguments to stdout
- mainly used in scripts, but also useful when exploring shell behaviour
- **echo** is often built in to shells for efficiency, but also provided by **/bin/echo**
- see also **/usr/bin/printf**
- Two useful **echo** options:

-n	do not output a trailing newline
-e	enable interpretation of backslash escapes (on by default in dash)

```
$ echo Hello Andrew
Hello Andrew
$ echo '\n'
\n
$ echo -e '\n'


$ echo -n Hello Andrew
Hello Andrew$
```


echo: implemented in Python

```
import sys
def main():
    """
    print arguments to stdout
    """
    print(' '.join(sys.argv[1:]))
```

source code for echo.py

echo: implemented in C

```c
// print arguments to stdout
int main(int argc, char *argv[]) {
    for (int i = 1; i < argc; i++) {
        if (i > 1) {
            fputc(' ', stdout);
        }
        fputs(argv[i], stdout);
    }
    fputc('\n', stdout);
    return 0;
}
```

source code for echo.c

Shell Variables

- shell variables are untyped - consider them as strings
  - note that **1** is equivalent to **"1"**
- shell variables are not declared
- shell variables do not need initialization
  - initial value is the empty string
- one scope - no local variables
  - except sub-shells & functions (sort-of)
  - changes to variables in sub-shells have no effect outside sub-shell
  - components of pipeline executed in sub-shell
- **`$name`** replaced with value of variable **`name`**
- **`name=value`** assigns **`value`** to variable **`name`**
  - note: no spaces around **=**

$(command) - command expansion:

- **$(*command*)** is evaluated by running ***command***
- stdout is captured from ***command***
- **$(*command*)** is replaced with the entire captured stdout
- **'*command*'** (backticks) is equivalent to **$(*command*)**
  - backticks is original syntax, so widely used
  - nesting of backticks is problematic

For example:

```
$ now=$(date)
$ echo $now
Sun 23 Jun 1912 02:31:00 GMT
$
```

' ' - Single Quotes

- single quotes **' '** group the characters within into a single word
  - no characters interpreted specially inside single quotes
  - variables, commands and arithmetic are not expanded inside single quotes
  - globbing and word-splitting does not occur inside double quotes
  - a single quote can not occur within single quotes
  - you can put a double quote between single-quotes

For example:

```
$ echo '***   !@#$%^&*(){}[]:;"<>?,./`   ***'
***   !@#$%^&*(){}[]:;"<>?,./`   ***
$ echo 'this is "normal"'
this is "normal"
```


"" - Double Quotes

- double quotes **""** group the characters within into a single word
  - variables, commands and arithmetic are expanded inside double quotes
  - backslash can be used to escape **$ “ “” ” **
  - other characters not interpreted specially inside double quotes
  - globbing and word-splitting does not occur inside double quotes
  - you can put a single quote between double-quotes

For example:

```
$ answer=42
$ echo "The answer is $answer."
The answer is 42.
$ echo 'The answer is $answer.'
The answer is $answer.
$ echo "time's up"
time's up
$ echo "*    *"
*    *
```


<< - here documents

- <<***word*** called a here document
- following lines until ***word*** specify multi-line string as command input
- variables and commands expanded - same as double quotes
- <<‘***word***’ variables and commands not expanded - same as single quotes
- <<-***word*** removes leading tabs from each line, allowing indentation within scripts

```
$ name=Andrew
$ tr a-z A-Z <<END-MARKER
Hello $name
How are you
Good bye
END-MARKER
HELLO ANDREW
HOW ARE YOU
GOOD BYE
```

Arithmetic

- **$((*expression*))** is evaluated as an arithmetic expression
    - ***expression*** is evaluated as C-like integer arithmetic
    - and is replaced with the result
    - the **$** on variables can be omitted in expressions
- shell arithmetic implementation slow compared to e.g. C
    - significant overhead converting to/from strings
- older scripts may use the separate program **`expr`** for arithmetic

For example:

```
$ x=8
$ answer=$((x*x - 3*x + 2))
$ echo $answer
42
```

- Note that variables in arithmetic expressions are recursively evaluated

word splitting

- coders not understanding how shells split words is a frequent source of bugs

```
# inspect how shell splits lines into program arguments (argv)
import sys
print(f'sys.argv = {sys.argv}')
```

source code for print_argv.py

```
$ v=''
$ ./print_argv.py $v
sys.argv = ['./print_argv.py']
$ ./print_argv.py "$v"
sys.argv = ['./print_argv.py', '']
$ w='   xx     yyy         zzzz    '
$ ./print_argv.py $w
sys.argv = ['./print_argv.py', 'xx', 'yyy', 'zzzz']
$ ./print_argv.py "$w"
sys.argv = ['./print_argv.py', '   xx     yyy         zzzz    ']
```


*?[]! - pathname globbing

- ***?[]!** characters cause a word to be matched against pathnames
    - confusingly similar to regexes - but much less powerful
- ***** matches 0 or more of **any** character - equivalent to regex **.***
- **?** matches any **one** characters - equivalent to regex **.**
- **[*characters*]** matches **1** of ***characters*** - same as regex **[]**
- **[!*characters*]** matches **1** character not in ***characters*** - same as regex **[^]**
- if no pathname matches the word is unchanged
- aside: globbing also available in Python, Perl, C & other languages

```
$ echo *.[ch]
functions.c functions.h i.h main.c
$ ./print_argv.py  *.[ch]
['./print_argv.py', 'functions.c', 'functions.h', 'i.h', 'main.c']
$ ./print_argv.py  '*.[ch]'
['./print_argv.py', '*.[ch]']
$ ./print_argv.py  "*.[ch]"
['./print_argv.py', '*.[ch]']
$ ./print_argv.py *.zzzzz
['./print_argv.py', '*.zzzzz']
```

I/O Redirection

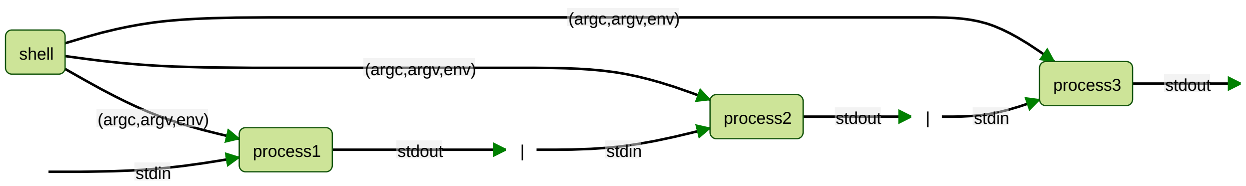
- stdin, stdout & stderr for a command can be directed to/from files

< *infile*	connect `stdin` to the file `infile`
> *outfile*	send `stdout` to the file `outfile`
>> *outfile*	append `stdout` to the file `outfile`
2> *outfile*	send `stderr` to the file `outfile`
2>> *outfile*	append `stderr` to the file `outfile`
> *outfile* 2>&1	send `stderr+stdout` to `outfile`
1>&2	send `stdout` to `stderr` (handy for error messages)
«*word*	here-document - previously discussed
«< *string*	(in bash) here-string - a single line here-document
&> *outfile*	(in bash) send `stdout+stderr` to `outfile`

- beware: > truncates file before executing command.
- always have backups!

Pipelines

- ***command*$_1$ | *command*$_2$ | *command*$_3$ | ...**
- stdout of ***command*$_{n-1}$** connected to stdin of ***command*$_n$**
- beware changes to variables in pipeline are lost
- some non-filter style Unix programs given a filename **–** read from stdin
  - allows them to be used in a pipeline



searching PATH for the program

- first word on line specifies command to be run
- if first word is not the full (absolute) pathname of a file the colon-separated list of directory specified by the variable PATH is searched
- for example if **`PATH=/bin/:/usr/bin/:/home/z1234567/bin`**
  and the command is **`kitten`** the shell will check (stat) these files in order:
  - **`/bin/kitten /usr/bin/kitten /home/z1234567/bin`**
  - the first that exists and is executable will be run
  - if none exist the shell will print an error message
- or **.** in PATH causes the current directory to be checked
  - this can be convenient - but make it last not first, e.g.:
    `PATH=/bin/:/usr/bin/:/home/z1234567/bin:.`
  - definitely do not include the current directory in PATH if you are root
  - an empty entry in PATH is equivalent to .

danger of having . in your PATH

- if . is not last in PATH then programs in the current directory may be unexpectedly run
- this can also happen inside run shell scripts or other programs you run
- robust shell scripts often set PATH to ensure this doesn't happen, e.g.: **PATH=/bin/:/usr/bin/:$PATH**

```
# equivalent to PATH=.:/bin:/usr/bin:/home/z1234567/bin
$ PATH=:/bin:/usr/bin:/home/z1234567/bin
$ cat >cat <<eof
#!/bin/dash
echo miaou
eof
$ chmod 755 cat
$ cat /home/cs2041/public_html/index.html
miaou
$
```

Problem: `./cat` is being run rather `/bin/cat`

Shell Scripts

We can execute shell commands in a file:

```
$ cat hello
echo Hello, John Connor - the time is $(date)
$ dash hello
Hello, John Connor - the time is Fri 29 Aug 1997 02:14:00 EST
```

- Unix-like systems allow an interpreter to be specified in a #! line
- allows program to be executed directly without knowing it is shell

```
$ cat hello
#!/usr/bin/env dash
echo Hello, John Connor - the time is $(date)
$ chmod 755 hello
$ ./hello
Hello, John Connor - the time is Fri 29 Aug 1997 02:14:00 EST
```

- use **#!/bin/bash** if you want bash

Shell Built-in Variables

Some shell built-in variables with pre-assigned values:

$0	the name of the command
$1	the first command-line argument
$2	the second command-line argument
...	...
$#	count of command-line arguments
"$@"	command-line arguments as separate word
$?	exit status of the most recent command
$$	process ID of this shell

- **$$** is useful for generating (somewhat) unique names in scripts.
- see also the **shift** command

Example - Shell Script using Built-in Variables

```
#!/bin/dash
# A simple shell script demonstrating access to arguments.
# written by andrewt@unsw.edu.au as a COMP(2041|9044) example
echo My name is "$0"
echo My process number is $$
echo I have $# arguments
echo My command-line arguments are "$@"
echo My 5th argument is "'$5'"
echo My 10th argument is "'${10}'"
```

source code for args.sh

Example - Simple Shell Script

```
#!/bin/sh
# l [file|directories...] - list files
#
# written by andrewt@unsw.edu.au as a COMP(2041|9044) example
#
# Short shell scripts can be used for convenience.
#
# It is common to put these scripts in a directory
# such as /home/z1234567/scripts
# then add this directory to PATH e.g in .bash_login
# PATH=$PATH:/home/z1234567/scripts
#
# Note:  "$@" expands to the arguments to the script,
# but preserves whitespace in arguments.
ls -las "$@"
```

source code for l

Example - Putting a Pipeline in a Shell Script

```
#!/bin/dash
# Count the number of time each different word occurs
# in the files given as arguments, or stdin if no arguments,
# e.g. word_frequency.sh dracula.txt
# written by andrewt@unsw.edu.au as a COMP(2041|9044) example
cat "$@" |                  # tr doesn't take filenames as arguments
tr 'A-Z' 'a-z'|             # map uppercase to lower case, better - tr '[:upper:
tr ' ' '\n' |               # convert to one word per line
tr -cd "a-z'" |             # remove all characters except a-z and '
grep -E -v '^$' |           # remove empty lines
sort |                      # place words in alphabetical order
uniq -c |                   # count how many times each word occurs
sort -rn                    # order in reverse frequency of occurrence
# notes:
# - first 2 tr commands could be combined
# - sed 's/ /\n/g' could be used instead of tr ' ' '\n'
# - sed "s/[^a-z']//g" could be used instead of tr -cd "a-z'"
```

source code for word_frequency.sh

Debugging Shell Scripts

Tip: debugging for shell scripts

- test parts of shell script from command line
- use **echo** to print the value of variables
- add **set -x** to see commands being executed
  - or equivalently run **/bin/dash -x script.sh**
  - shell transforms commands
  - useful to see exactly what is being executed

Exit Status and Control

- when Unix-like programs finish they give the operating system an **exit status**
  - the return value of ‘main becomes the **exit status** of a C program
  - or if `exit` is called, its argument is the **exit status**
  - in Python **exit status** is supplied as an argument to `sys.exit`
- an **exit status** is a (usually small) integer
  - by convention a zero exit status indicated normal/successful execution
  - a non-zero exit status indicates an error occurred
  - which non-zero integer might indicate the nature of the problem
- program exit status is often ignored
  - not important writing single programs (COMP1511/COMP9021)
  - very important when combining multiple programs COMP(2041|9044)
- flow of execution in Shell scripts based on exit status
  - `if`/`while` statement conditions use exit status
- two weird utilities
  - `/bin/true` does nothing and always exits with status 0
  - `/bin/false` does nothing and always exits with status 1

The `test` command

- The **test** command performs a test or combination of tests and:
  - does/prints nothing
  - returns a zero exit status if the test succeeds
  - returns a non-zero exit status if the test fails
- Provides a variety of useful operators:
  - string comparison: **= !=**
  - numeric comparison: **-eq -ne -lt**
  - test if file exists/is executable/is readable: **-f -x -r**
  - boolean operators (and/or/not): **-a -o !**
- also available as ‘[’ instead of test - which many programmers prefer
- builtin to some shell (e.g. bash) but available as **/bin/test** or **/bin/[**

The test command examples

```
# does the variable msg have the value "Hello"?
test "$msg" = "Hello"

# does x contain a numeric value larger than y?
test "$x" -gt "$y"

# Error: expands to "test hello there = Hello"?
msg="hello there"
test $msg = Hello

# is the value of x in range 10..20?
test "$x" -ge 10 -a "$x" -le 20

# is the file xyz a readable directory?
test -r xyz -a -d xyz

# alternative syntax; requires closing ]
[ -r xyz -a -d xyz ]
```


If Statements - syntax

```
if command1
then
    then-commands
elif command2
then
    elif-commands
else
    else-commands
fi
```

- the execution path depends on the exit status of *command*$_1$ and *command*$_2$
- ***command1*** is executed and if its exit status is 0, the ***then-commands*** are executed
- otherwise ***command2*** is executed and if its exit status is 0, the ***elif-commands*** are executed
- otherwise the ***else-commands*** are executed

If Statements - Example

```
if gcc main.c; then
    echo your C compiles
elif python3 main.c; then
    echo you have written Python not C
else
    echo program broken - send help
fi
```

```
if gcc a.c
then
# you can not have an empty body
# use a : statement which does nothing
    :
else
    rm a.c
fi
```

While Statements - syntax

shell **while** statements have this form:

```
while command
do
   body-commands
done
```

- the execution path depends on the exit status of *command*
- ***command*** is executed and if its exit status is 0,
  the ***body-commands*** are executed
  and then ***command*** is executed and if its exit status is 0
  the ***body-commands*** are executed
  and ...
- if the exit status of ***command~*** is not 0, execution of the loop stops

example - seq - simple version

```
#!/bin/dash
# simple emulation of /usr/bin/seq for a COMP(2041|9044) example
# andrewt@unsw.edu.au
# Print the integers 1..n with no argument checking
last=$1
number=1
while test $number -le "$last"
do
    echo $number
    number=$((number + 1))
done
```

source code for seq.v0.sh

```
$ ./seq.v0.sh 3
1
2
3
```


example - seq - argument handling added

```
# Print the integers 1..n or n..m
if test $# = 1
then
    first=1
    last=$1
elif test $# = 2
then
    first=$1
    last=$2
else
    echo "Usage: $0 <last> or  $0 <first> <last>" 1>&2
    exit 1
fi
number=$first
while test $number -le "$last"
do
    echo $number
    number=$((number + 1))
done
```

source code for seq.v1.sh

example - seq - using [] instead of test

```
if [ $# = 1 ]
then
    first=1
    last=$1
elif [ $# = 1 ]
then
    first=$1
    last=$2
else
    echo "Usage: $0 <last> or  $0 <first> <last>" 1>&2
    exit 1
fi
number=$first
while [ $number -le $last ]
do
    echo $number
    number=$((number + 1))
done
```

source code for seq.v2.sh

example - watching a website - argument checking

```
# Repeatedly download a specified web page
# until a specified regexp matches its source
# then notify the specified email address.
#
# For example:
# watch_website.sh http://ticketek.com.au/ '[Tt]ayl(a|or) *[Ss]wift' andrewt@unsw
repeat_seconds=300  #check every 5 minutes
if test $# = 3
then
    url=$1
    regexp=$2
    email_address=$3
else
    echo "Usage: $0 <url> <regex> <email-address>" 1>&2
    exit 1
fi
```

source code for watch_website.sh

example - watching a website - main loop

```
while true
do
    if curl --silent "$url"|grep -E "$regexp" >/dev/null
    then
        # the 2nd echo is for testing, remove to really send email
        echo "Generated by $0" |
        echo mail -s "website '$url' now matches regex '$regexp'" "$email_address
        exit 0
    fi
    sleep $repeat_seconds
done
```

source code for watch_website.sh

For Statements in Shell

shell **for** statements have this form:

```
for var in word1 word2 word3
do
    body-commands
    ...
done
```

- the loop executes once for each *word* with *var* set to the *word*
- **break** & **continue** statements can be in used inside for & while loops with the same effect as C/Python
- keywords such **for**, **if**, **while**, ... are only recognised at the start of a command, e.g.:

```
$ echo when if else for
when if else for
```


Example - Shell Script accessing Command-line Arguments

```
  echo "$a"
done
```

source code for accessing_args.sh

Example - Shell Script accessing Command-line Arguments

```
$ ./accessing_args.sh one two "three four"
one
two
three four
```


Using Exit Status for Conditional Execution

- all commands are executed if separated by **;** or newline, e.g:
  cmd_1 **;** cmd_2 **;** ... **;** cmd_n
- when commands are separated by **&&**
  cmd_1 **&&** cmd_2 **&&** ... **&&** cmd_n
  execution stops if a command has non-zero exit status
  cmd_{n+1} is executed only if cmd_n has zero exit status
- when commands are separated by **||**
  cmd_1 **||** cmd_2 **||** ... **||** cmd_n
  execution stops if a command haszero exit status
  cmd_{n+1} is executed only if cmd_n has non-zero exit status
- **{}** can be used to group commands
- **()** also can be used to group commands - but executes them in a subshell
  - changes to variables and current working directory have no effect outside the subshell
- exit status of group or pipeline of commands is exit status of last command

Conditional Execution Examples

```
# run a.out if it exists and is executablr
test -x a.out && ./a.out

# if directory tmp doesn't exist create it
test -d tmp || mkdir tmp

# if directory tmp doesn't exist create it
{test -d tmp || mkdir tmp;} && chmod 755 tmp

# but simpler is
mkdir -p tmp && chmod 755 tmp
```


{} versus () - example

```
$ cd /usr/share
$ x=123
$ ( cd /tmp;  x=abc; )
$ echo $x
123
$ pwd
/usr/share
$ { cd /tmp;  x=abc; }
$ echo $x
abd
$ pwd
/tmp
```

- changes to variables and current working directory have no effect outside a subshell
- pipelines also executed in subshell, but variables and directory not usually changed in a pipeline

shellcheck - shell static analysis tool

- shellcheck https://www.shellcheck.net/ statically analyzes shell scripts
  - finds possible bugs without running script
  - highly-recommended because it picks up many common shell coding mistakes
- static analysis tools higly valuable because they give another way of checking for errors
  - faster/easier than testing
  - may find errors testing will miss
- static analysis tools available for many languages
  - e.g. pyflakes, pylint, prospector for Python
  - compilers (e.g. gcc/clang) use static analysis to produce faster/smaller code and report possible bugs

example - renaming files - argument checking

```
# Change the names of the specified files to lower case.
# (simple version of the perl utility rename)
#
# Note use of test to check if the new filename is unchanged.
#
# Note the double quotes around $filename so filenames
# containing spaces are not broken into multiple words
# Note the use of mv -- to stop mv interpreting a
# filename beginning with - as an option
# Note files named -n or -e still break the script
# because echo will treat them as an option,
if test $# = 0
then
    echo "Usage $0: <files>" 1>&2
    exit 1
fi
```

source code for tolower.sh

example - renaming files- main loop

```
for filename in "$@"
do
    new_filename=$(
        echo "$filename"|
        tr '[:upper:]' '[:lower:]'
        )
    test "$filename" = "$new_filename" &&
        continue
    if test -r "$new_filename"
    then
        echo "$0: $new_filename exists" 1>&2
    elif test -e "$filename"
    then
        mv -- "$filename" "$new_filename"
    else
        echo "$0: $filename not found" 1>&2
    fi
done
```

source code for tolower.sh

read - shell builtin

- **read** is a shell builtin which reads a line of input into variables(s)
  - non-zero exit status on EOF
  - newline is stripped
  - leading and trailing whitespace stripped unless variable IFS unset
  - note **-r** option if input might contains backslashes
- if more than one variable specified, line is split into fields on white space
  - 1st variable assigned 1st field, 2nd variable assigned 2nd field ...
  - last variable entire remainder of line
  - if insufficient fields variables assigned empty strings
- if more than one variable specified, line is split into fields on white space

```
$ read v
hello world
$ echo "$v"
hello world
$ read a b c
1 2 3 4 5
$ echo "a='$a' b='$b' c='$c'"
a='1' b='2' c='3 4 5'
```

read - simple example

```
echo -n "Do you like learning Shell? "
read answer
# get first letter of answer connverted to lower case
answer="$(
    echo "$answer"|
    cut -c1|
    tr A-Z a-z
    )"
if test "$answer" = "y"
then
    response=":)"
elif test "$answer" = "n"
then
    response=":("
else
    response="??"
fi
echo "$response"
```

source code for read_response_if.sh

emulating cat with read

```
#!/bin/dash
# written by andrewt@unsw.edu.au for COMP(2041|9044)
# over-simple /bin/cat emulation using read
# setting the special variable IFS to the empty string
# stops trailing white space being stripped
for file in "$@"
do
    while IFS= read -r line
    do
        echo "$line"
    done <$file
done
```

source code for read_cat.sh

case statements - syntax

```
case word in
pattern1)
    commands1
    ;;
pattern2)
    commands2
    ;;
patternn)
    commandsN
esac
```

- ***word*** is compared to each $\textbf{\textit{pattern}}_i$ in turn.
- for the first $\textbf{\textit{pattern}}_i$ that matches
  the corresponding $\textbf{\textit{commands}}_i$ is executed
  and the case statement finishes.

case statements in Shell

- case patterns use the same language as filename expansion (globbing)
  - in other words the special characters are `*` `?` `[]`
  - patterns are not interpreted as regexes
- shell programmer used to use **case** statements heavily for efficiency
  - much less important now and many shell programmers don't use `case`
  - but use of `case` can still make shell code more readable

case statement - examples

```
# Checking number of command line args
case $# in
0)  echo "You forgot to supply the argument" ;;
1)  filename=$1 ;;
*)  echo "You supplied too many arguments" ;;
esac

# Classifying a file via its name
case "$file" in
*.c)  echo "$file looks like a C source-code file" ;;
*.h)  echo "$file looks like a C header file" ;;
*.o)  echo "$file looks like a an object file" ;;
...
?)    echo "$file's name is too short to classify" ;;
*)    echo "I have no idea what $file is" ;;
esac
```


case - simple example

```
echo -n "Do you like learning Shell? "
read answer
case "$answer" in
[Yy]*)
    response=":)"
    ;;
[Nn]*)
    response=":("
    ;;
*)
    response="??"
esac
echo "$response"
```

source code for read_response_case.sh

creating a 1001 file C program - getting started

```
# this program creates 1000 files f0.c .. f999.c
# file f$i.c contains function f$i which returns $i
# for example file42.c contains function f42 which returns 42
# main.c is created with code to call all 1000 functions
# and print the sum of their return values
#
# first add the initial lines to main.c
# note the use of quotes on eof to disable variable interpolation
# in the here document
cat >main.c <<'eof'
#include <stdio.h>
int main(void) {
    int v = 0 ;
eof
```

source code for create_1001_file_C_program.sh

creating a 1001 file C program - creating the files

```
i=0
while test $i -lt 1000
do
    # add a line to main.c to call the function f$i
    cat >>main.c <<eof
    int f$i(void);
    v += f$i();
eof
    # create file$i.c containing function f$i
    cat >file$i.c <<eof
int f$i(void) {
    return $i;
}
eof
    i=$((i + 1))
done
```

source code for create_1001_file_C_program.sh

creating a 1001 file C program - compiling & running the program

```
cat >>main.c <<'eof'
    printf("%d\n", v);
    return 0;
}
eof
# compile and run the 1001 C files
time clang main.c file*.c
./a.out
```

source code for create_1001_file_C_program.sh

shell functions

shell functions have this form:

```
name () {
    commands
}
```

- function arguments passed in: **`$@ $1 $2`** ...
- use **`return`** to stop function execution and return exit status
  - beware: **`exit`** in a function still terminates entire program
- **`local`** keyword can be used to limit scope of variables to function
  - **`local`** is not POSIX, but is widely supported although exact semantics vary
    - **ksh** does not support **`local`**, it has a similar keyword **`typeset`**

example - shell function

```
#!/bin/dash
# written by andrewt@unsw.edu.au for COMP(2041|9044)
# demonstrate simple use of a shell function
favourite_command() {
    name=$1
    command=$2
    echo "My name is $name, my favourite Unix command is $command."
}
favourite_command Andrew "uniq"
favourite_command Dylan "jq"
favourite_command Grace "sed"
```

source code for favourite_command.sh

example - local variables in a shell function

```
# print print numbers < 1000
# note use of local Shell builtin to scope a variable
# without the local declaration
# the variable i in the function would be global
# and would break the bottom while loop
# local is not (yet) POSIX but is widely supported
is_prime() {
    local n i
    n=$1
    i=2
    while test $i -lt $n
    do
        test $((n % i)) -eq 0 &&
            return 1
        i=$((i + 1))
    done
    return 0
}
i=0
while test $i -lt 1000
do
    is_prime $i &&
        echo $i
    i=$((i + 1))
done
```

source code for local.sh

example plagiarism detection - simple diff

```
# Note use of diff -iw so changes in white-space or case are ignored
for file1 in "$@"
do
    for file2 in "$@"
    do
        test "$file1" = "$file2" &&
            break # avoid comparing pairs of assignments twice
        if diff -iBw "$file1" "$file2" >/dev/null
        then
            echo "$file1 is a copy of $file2"
        fi
    done
done
```

source code for plagiarism_detection.simple_diff.sh

plagiarism detection - ignoring changes to comments

```
# The substitution s/\/\/.*// removes // style C comments.
# This means changes in comments won't affect comparisons.
# Note use of temporary files is insecure - an attacker can anticipate the filename
TMP_FILE1=/tmp/plagiarism_tmp1$$
TMP_FILE2=/tmp/plagiarism_tmp2$$
for file1 in "$@"
do
    for file2 in "$@"
    do
        test "$file1" = "$file2" &&
            break # avoid comparing pairs of assignments twice
        sed 's/\/\/.*//' "$file1" >$TMP_FILE1
        sed 's/\/\/.*//' "$file2" >$TMP_FILE2
        if diff -i -w $TMP_FILE1 $TMP_FILE2 >/dev/null
        then
            echo "$file1 is a copy of $file2"
        fi
    done
done
rm -f $TMP_FILE1 $TMP_FILE2
```

source code for plagiarism_detection.comments.sh

robust creation & removal of temporary files

- our code can be more robust and more secure by using mktemp to generate temporary file names
- we can also use the builtin shell **`trap`** command to ensure temporary files are removed however the script exits
- temporary file creation is major source of security holes be very careful creating temporary files
- in all languages, use existing robust & well-tested code such as **`mktemp`**
  - don't write your own code
- `mktemp` is not (yet) standardized by POSIX
  - simple uses are portable to many platforms

plagiarism detection - ignoring changes to variable names #1

```
# change all C strings to the letter 's'
# and change all identifiers to the letter 'v'.
# Hence changes in strings & identifiers will be ignored.
# mktemp provide suitable temporary filename, robustly & securely
TMP_FILE1=$(mktemp)
TMP_FILE2=$(mktemp)
# trap allows use to remove temporary files if program interrupted
trap 'rm -f $TMP_FILE1 $TMP_FILE2' EXIT
# s/"["]*"/s/g changes strings to the letter 's'
# It won't match a few C strings which is OK for our purposes
# s/[a-zA-Z_][a-zA-Z0-9_]*/v/g changes variable names to 'v'
# It will also change function names, keywords etc. which is OK for our purposes.
transform() {
    sed '
        s/\/\/.*//
        s/"[^"]"/s/g
        s/[a-zA-Z_][a-zA-Z0-9_]*/v/g
        ' $1
}
```

source code for plagiarism_detection.identifiers.sh

plagiarism detection - ignoring changes to variable names #2

```
for file1 in "$@"
do
    for file2 in "$@"
    do
        test "$file1" = "$file2" &&
            break # avoid comparing pairs of assignments twice
        transform "$file1" >$TMP_FILE1
        transform "$file2" >$TMP_FILE2
        if diff -iBw $TMP_FILE1 $TMP_FILE2 >/dev/null
        then
            echo "$file1 is a copy of $file2"
        fi
    done
done
```

source code for plagiarism_detection.identifiers.sh

plagiarism detection - ignoring changes in code order

```
TMP_FILE1=$(mktemp)
TMP_FILE2=$(mktemp)
trap 'rm -f $TMP_FILE1 $TMP_FILE2' EXIT
# Note the use of sort so line reordering won't prevent detection of plagiarism.
transform() {
    sed '
        s/\/\/.*//
        s/"[^"]"/s/g
        s/[a-zA-Z_][a-zA-Z0-9_]*/v/g
        ' $1|
    sort
}
```

source code for plagiarism_detection.reordering.sh

Example - creating a temporary directory

```
# securely & robustly create a new temporary directory
temporary_directory=$(mktemp -d)
# ensure temporary directory + all its contents removed on exit
trap 'rm -rf "$temporary_directory"; exit' INT TERM EXIT
# change working directory to the new temporary directory
cd "$temporary_directory" || exit 1
# we are now in an empty directory
# and create any number of files & directories
# which all will be removed by the trap above
# e.g. create one thousand empty files
seq 1 1000|xargs touch
# print current directory and list files
pwd
ls -l
```

source code for create_temporary_directory.sh

Cryptographic hash function

- algorithm maps byte sequence of any length to certain number of bits
- e.g sha256 input: any number of bytes, output 256 bits (= 8 bytes) hash
- one way function - not feasible to reverse
- given a hash, not feasible to compute an input which produces that hash
- collisions (different inputs producing the same hash) occur but are vanishingly rare
- small change to input changes hash completely
- many applications:
    - hashes of passwords stored rather than password itself
    - integrity check on set of files
    - fingerprint a file

plagiarism detection - using hashing

```
# Improved version of plagiarism_detection.reordering.sh
# Note use sha256sum to calculate a Cryptographic hash of the modified file
# https://en.wikipedia.org/wiki/SHA-2
# and  use of sort && uniq to find files with the same hash
# This allows execution time linear in the number of files
# We could use a faster less secure hashing function instead of sha2
sha2hash() {
    sed '
        s/\/\/.*//
        s/"[^"]"/s/g
        s/[a-zA-Z_][a-zA-Z0-9_]*/v/g
        ' $1|
    sort|
    sha256sum
}
for file in "$@"
do
    echo "$(sha2hash $file) $file"
done|
sort|
uniq -w32 -d --all-repeated=separate
```

source code for plagiarism_detection.hash.sh

example - using a signal to provide a time limit

```
my_process_id=$$
# launch a asynchronous sub-shell that will kill
# this process in a second
(sleep 1; kill $my_process_id) &
i=0
while true
do
    echo $i
    i=$((i + 1))
done
```

source code for async.v0.sh

- ***command* &** executes *command* but does not wait for it to finish
- **sleep 1** suspends execution for a second
- **kill** sends a signal to a process, which by default causes it to exit

intercepting signals with trap

- **trap** specifies commands to be executed if a signal is received, e.g.:

```
# count slowly and laugh at interrupts (ctrl-C)
# catch signal SIGINT and  print message
trap 'echo ha ha' INT
n=0
while true
do
    echo "$n"
    sleep 1
    n=$((n + 1))
done
```

source code for laugh.sh

- **trap** is useful for cleaning up temporary files before termination, e.g.

```
    trap 'rm -f "$TMP_FILE";exit' INT TERM EXIT
```


example - catching a signal with trap

```
# catch signal SIGTERM, print message and exit
trap 'echo loop executed $n times in 1 second; exit 0' TERM
# launch a sub-shell that will terminate
# this process in 1 second
my_process_id=$$
(sleep 1; kill $my_process_id) &
n=0
while true
do
    n=$((n + 1))
done
```

source code for async.v1.sh

example - compiling in parallel

```
# compile the files of a muti-file C program in parallel
# use create_1001_file_C_program.sh to create suitable test data
# On a CPU with n cores this can be (nearly) n times faster
# If there are large number of C files we
# may exhaust memory or operating system resources
for f in "$@"
do
    clang -c "$f" &
done
# wait for the incremental compiles to finish
# and then compile .o files into single binary
wait
clang -o binary -- *.o
```

source code for parallel_compile.v0.sh

example - compiling in parallel

```
# compile the files of a muti-file C program in parallel
# use create_1001_file_C_program.sh to create suitable test data
# on Linux getconf will tell us how many cores the machine has
# otherwise assume 8
max_processes=$(getconf _NPROCESSORS_ONLN 2>/dev/null) ||
    max_processes=8
# NOTE: this breaks if a filename contains whitespace or quotes
echo "$@"|
xargs --max-procs=$max_processes --max-args=1 clang -c
clang -o binary -- *.o
```

source code for parallel_compile.v1.sh

example - compiling in parallel

```
$ ./create_1001_file_C_program.sh
$ echo *.c
file0.c file1.c file10.c file100.c file101.c file102.c ...
$ echo *.c|wc -w
1001
# compiling 1 file at a time
$ time clang *.c
real    0m20.875s
user    0m13.016s
sys     0m7.835s
# compiling all 1001 files simultaneously
$ time ./parallel_compile.v0.sh *.c
real    0m2.335s
user    0m9.066s
sys     0m8.788s
# compiling 24 files at time
$ time ./parallel_compile.v1.sh *.c
real    0m1.971s
user    0m18.694s
sys     0m18.428s
$ grep 'model name' /proc/cpuinfo|sed 1q
model name  : AMD Ryzen 9 3900X 12-Core Processor
```

example - compiling in parallel

```
# compile the files of a multi-file C program in parallel
# use create_1001_file_C_program.sh to create suitable test data
# find's -print0 option terminates pathnames with a '\0'
# xargs's --null option expects '\0' terminated input
# as '\0' can not appear in file names this can handle any filename
# on Linux getconf will tell us how many cores the machine has
# if getconf assume 8
max_processes=$(getconf _NPROCESSORS_ONLN 2>/dev/null) ||
    max_processes=8
find "$@" -print0|
xargs --max-procs=$max_processes --max-args=1  --null clang -c
clang -o binary -- *.o
```

source code for parallel_compile.v2.sh

example - compiling in parallel

```
# compile the files of a muti-file C program in parallel
# use create_1001_file_C_program.sh to create suitable test data
parallel clang -c '{}' ::: "$@"
clang -o binary -- *.o
```

source code for parallel_compile.v3.sh

Shell Variable Expansion - More Syntax

```
$ x=1
$ y=fred
$ echo $x$y
1fred
$ echo $xy        # the aim is to display "1y"

$ echo "$x"y
1y
$ echo ${x}y
1y
$ echo ${j-10}    # give value of j or 10 if no value
10
$ echo ${j=33}    # set j to 33 if no value (and give $j)
33
$ echo ${x:?No Value}   # display "No Value" if $x not set
1
$ echo ${xx:?No Value}  # display "No Value" if $xx not set
-bash: xx: No Value
```

Bash arithmetic (()) extension example

```
# print print numbers < 1000
# Rewritten to use bash arithmetic extension (())
# This makes the program more readable but less portable.
is_prime() {
    local n i
    n=$1
    i=2
    while ((i < n))
    do
        if ((n % i == 0))
        then
            return 1
        fi
        i=$((i + 1))
    done
    return 0
}
i=0
while ((i < 1000))
do
    is_prime $i && echo $i
    i=$((i + 1))
done
```

source code for bash_arithmetic.sh