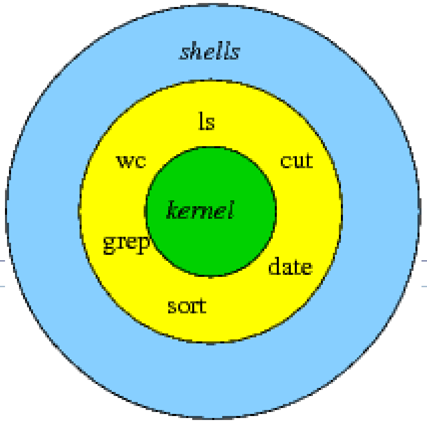
COMP 2021

Unix and Script Programming



Shell Programming

Shells and Script

- A shell can be used in one of two ways:
  - A *command interpreter*, used interactively
  - A *programming language*, to write shell scripts (your own custom commands)

- Script
  - *It is very similar to a program, although it is usually much simpler to write and it is executed from source code (or byte code) via an interpreter. Shell scripts are scripts designed to run within a command shell.*

Practical Example

- Distribute grade to individual student via email

```
#!/bin/sh
# read csv file and distribute student grade via email

input="SampleGrade.csv"
# set "," as the field separator using $IFS
# and read line by line using while loop

subject="COMP2021 HW1 grade"

while IFS="," read -r stuid email grade
do
  echo "$stuid $email $grade"
  echo "Dear $stuid Your COMP2021 HW1 grade is $grade" | tr -d \\r | mail -s "$subject" $email
  echo "email sent!"
done < "$input"
```

SampleGrade.csv

```
1,aaa@ust.hk,95
2,bbb@ust.hk,87
3,lixin@ust.hk,100
```

Shell Scripts

- A shell script is just a file containing shell commands, but with a few extras:
  - The first line of a shell script should begin with a shebang (#!), followed by the full path of the shell we'd like to use as an interpreter:

    ```
    #!/bin/sh
    ```

    for a most commonly used Bourne shell script.
  - A shell script must be readable and executable.

    ```
    chmod u+rx scriptname
    ```

  - As with any command, a shell script has to be "in your path" to be executed.
    - If "." is not in your PATH, you must specify "`./scriptname`" instead of just "`scriptname`"

Shell Script Example

"Hello World"

- Here is a “hello world” shell script:

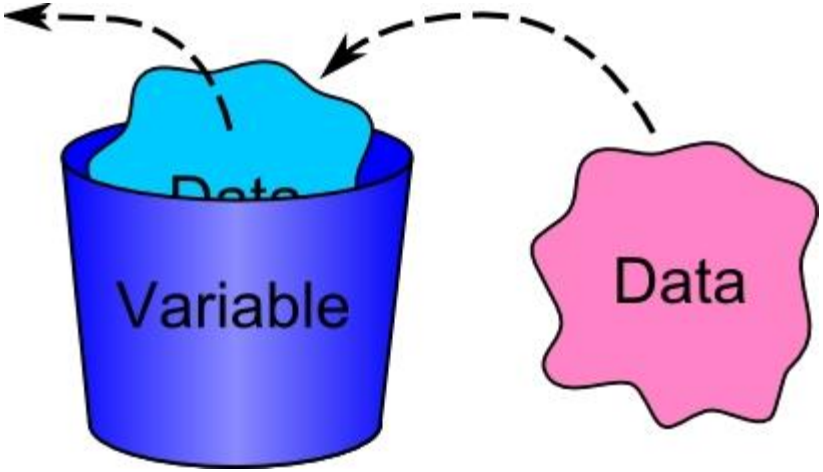
```
$ ls -l
-rwxr-xr-x  1 cindy   48 Feb 19 11:50 HelloWorld
$ cat helloworld.sh
#!/bin/sh
# comment lines start with the # character
echo "Hello world"
$ helloworld.sh
Hello World
```

- The `echo` command functions like a print command in shell scripts.

Shell Variables

- To get anything done we need variables
- To read the values in variables, precede their names by a `$`
- The contents of any variable can be listed using the `echo` command
- Types of variables: local and environment

```
$ echo $SHELL
/bin/tcsh
```



Shell Variables (Cont.)

- The user variable name can be any sequence of letters, digits, and the underscore character, but the first character must be a letter
- Internally, all values are stored as strings.

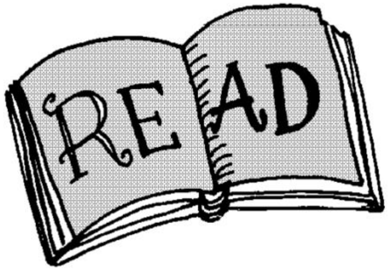
```
$ cat variable.sh
#! /bin/sh
# There cannot be any space before or after the "="
# Internally, all values are stored as strings

number=50
course="COMP2021"
echo "$course has $number students"
$ variable.sh
COMP2021 has 50 students
```

User Input

- Use the `read` command to get and store input from the user.

```
$ cat read.sh
#!/bin/sh
# Use read command to get and store input from the user
echo "Enter name: "
read name
echo "How many girlfriends do you have?"
read number
echo "$name has $number girlfriends!"
$ read.sh
Enter name:
Hoffman Playboy
How many girlfriends do you have?
too many
Hoffman Playboy has too many girlfriends!
```



User Input (Cont.)

- `read` reads one line of input from the keyboard and assigns it to one or more user-supplied variables.
- Leftover input words are all assigned to the last variable.

```
$ cat read2.sh
#!/bin/sh
# Use read command to get and store input from the user
echo "Enter name and how many girlfriends: "
read name number
echo "$name has $number girlfriends!"
$ read2.sh
Enter name and how many girlfriends:
Edison Chen 50
Edison has Chen 50 girlfriends!
$ read2.sh
Enter name and how many girlfriends:
Poorguy
Poorguy has  girlfriends!
```

Quoting

Quoting	Description
Single quote	All special characters between these quotes lose their special meaning.
Double quote	Most special characters between these quotes lose their special meaning with these exceptions: `` $,  `,  \$,  \‘,  \“,  \\ ``
Backslash	Any character immediately following the backslash loses its special meaning
Back Quote	Anything in between back quotes would be treated as a command and would be executed

Quoting (Cont.)

```
$ cat quoting.sh
#!/bin/sh
# Different quotes
DATE=`date`
echo "Current date: $DATE“
user=`whoami`
numusers=`who | wc -l`
echo "Hi $user! There are $numusers users logged on."
echo "I have \$5000.“
echo "It\'s Shell Programming"
echo '<-$1500.**>; (update?) [y\n]‘
$ quoting.sh
Current date: Sat Feb 27 10:27:51 HKT 2016
Hi Cindy! There are 5 users logged on.
I have $5000.
It\'s Shell Programming
<-$1500.**>; (update?) [y\n]
```

`expr`

- Shell programming is not good at numerical computation, it is good at text processing.
- `expr` command allows simple *integer* calculations.
  - `+, -, \*, /, %, =, ==, !=`
- Here is an interactive Bourne shell example:

```
$ i=1
$ expr $i + 1
2
```

- To assign the result of an `expr` command to another shell variable, surround it with backquotes:

```
$ i=1
$ i=`expr $i + 1`
$ echo $i
2
```

expr (cont.)

- The * character normally means “all the files in the current directory”, so you need a “\” to use it for multiplication:

```
$ i=2
$ i=`expr $i \* 3`
$ echo $i
6
```

- expr also allows you to group expressions, but the “(“ and “)” characters also need to be preceded by backslashes:

```
$ i=2
$ echo `expr 5 \* \( $i + 3 \)`
25
```

expr Example

```
$ cat expr.sh
#!/bin/sh
# Example of expr, a simple calculator

echo "Enter the first operand: "
read a
echo "Enter the second operand: "
read b
echo "$a + $b = `expr $a + $b`"
x=`expr $a - $b`
echo "$a - $b = $x"
y=`expr $a \* $b`
echo "$a * $b = $y"
echo "$a / $b = `expr $a / $b`"
```

Control Flow

- The shell allows several control flow statements:
  - `if`
  - `while`
  - `for`
  - `break`

if

- The `if` statement works mostly as expected:
  - `if then fi`
  - `if then else fi`
  - `if then elif then else fi`

```
$ cat if_greeting.sh
#!/bin/sh
user=`whoami`
if [ $user = “cindy" ]
then
  echo "Hi cindy!"
fi
$ if_greeting.sh
Hi cindy!
```

- the spaces before and after the square brackets [ ] are required.

`if` Example 1

- The `if then else` statement

```
$ cat if_evenodd.sh
#!/bin/sh
echo "Enter the number:"
read n
num=$(expr $n % 2)
if [ $num -eq 0 ]
then
      echo "is a even number."
else
      echo "is an odd number."
fi
```

if Example 2

- The if then elif else statement

```
$ cat if_load.sh
#!/bin/sh
users=`who | wc -l`
if [ $users -ge 4 ]
then
       echo "Heavy load"
elif [ $users -gt 1 ]
then
      echo "Medium load"
else
      echo "Just me!"
fi
$ if_load.sh
Just me!
```

while Example: Factorial

```
$ cat while_factorial.sh
#!/bin/sh
# use while to do factorial
echo "Enter member: "
read n
fac=1
i=1
while [ $i -le $n ]
do
      fac=`expr $fac \* $i`
      i=`expr $i + 1`
done
echo "The factorial of $n is $fac"
$ while_factorial.sh
Enter number:
5
The factorial of 5 is 120
```

while Example 2: Armstrong

```
$cat while_armstrong.sh
#!/bin/sh
echo "Enter a number"
read n
arm=0
temp=$n
while [ $temp -ne 0 ]
do
  r=$(expr $temp % 10)
  arm=$(expr $arm + $r \* $r \* $r)
  temp=$(expr $temp / 10)
done
echo "Number is $n, cubes of its digits is $arm"
if [ $arm -eq $n ]
then
  echo "Armstrong"
else
  echo "Not Armstrong"
fi
```

371 is an Armstrong number, since

33 + 7**3 + 1**3 = 371**

break Example

- The break command works like in C++, breaking out of the innermost loop

```
$ cat while_break.sh
#!/bin/sh
while [ 1 ]
do
      echo "Wakeup [yes/no]?"
      read resp
      if [ $resp = "yes" ]
      then
              break
      fi
done
$ while_break.sh
Wakeup [yes/no]?
no
Wakeup [yes/no]?
y
Wakeup [yes/no]?
yes
```

Boolean Expressions

- Relational operators:

`-eq, -ne, -gt, -ge, -lt, -le`

- File operators:

`-f` *file*	True if *file* exists and is not a directory
`-d` *file*	True if *file* exists and *is* a directory
`-s` *file*	True if *file* exists and has a size > 0

- String operators:

`-z` *string*	True if the length of *string* is zero
`-n` *string*	True if the length of *string* is nonzero
s1 = *s2*	True if *s1* and *s2* are the same
s1 != *s2*	True if *s1* and *s2* are different
s1	True if *s1* is not the null string

- Boolean operators:

`!, -a, -o` (or `&& ||`)

Environment Variables

- An environment variable is a variable that is available to any child process of the shell.

  You can use (and change) them.

`HOME`	The path to your home directory
`PATH`	Directories where the shell looks for executables
`USER`	Your login name
`SHELL`	The name of the shell you are running
`PWD`	The current working directory

Environment Variable Example

```
$ cat env_variable.sh
#!/bin/sh
echo "Hi $USER!"
echo "Your home directory: $HOME"
echo "Your path: $PATH"
echo "Your current directory: $PWD"
echo "Your shell: $SHELL"

echo "The list of all environment variables"
echo `env`
```

Command Line Arguments

- The command line arguments that you call a script with are stored in variables `$1, $2, ..., $9` (positional parameters).

```
$ cat arguments.sh
#!/bin/sh
echo "The arguments are $1 $2 $3 $4 $5 $6 $7 $8 $9‟
echo “There’re $# arguments‟

$ arguments.sh a1 a2 a3 a4 a5 a6 a7 a8 a9 a10
The arguments are a1 a2 a3 a4 a5 a6 a7 a8 a9
There’re 10 arguments
```

- With more than 9 arguments, they are still stored, but they have to be moved using the `shift` command before they can be accessed.
- `$#` is the number of arguments received

Command Line Argument Example

- A script to swap two files

```
$ cat swapfile.sh
#!/bin/sh
if [ -f $1 ] && [ -f $2 ]
then
     mv $1 /tmp/$1
     mv $2 $1
     mv /tmp/$1 $2
else
     echo "file doesn't exist!"
fi
$ cat file1
This is file1
$ cat file2
This is file2
$ swapfile.sh file1 file2
$ cat file1
This is file2
$ cat file2
This is file1
```

shift

- The `shift` command promotes each command line argument by one (e.g., the value in `$2` moves to `$1`, `$3` moves to `$2`, etc.)

```
$ cat shiftargs.sh
#!/bin/sh
echo "The arguments are 0 = $0, 1 = $1, 2 = $2"
shift
echo "The arguments are 0 = $0, 1 = $1, 2 = $2"
shift
echo "The arguments are 0 = $0, 1 = $1, 2 = $2"
$ shiftargs.sh arg1 arg2 arg3
The args are 0 = shiftargs, 1 = arg1, 2 = arg2
The args are 0 = shiftargs, 1 = arg2, 2 = arg3
The args are 0 = shiftargs, 1 = arg3, 2 =
```

- $0 is the name the user typed to invoke the shell script
- The previous `$1`, `$2` becomes inaccessible

shift Example

- A general version of the swap command for two or more files?

```
swap f1 f2 f3 ... fn_1 fn

  f1   <--- f2
  f2   <--- f3
  f3   <--- f4
  ...
  fn_1 <--- fn
  fn   <--- f1
```

```
$cat swapmanyfiles.sh

#!/bin/sh
orig1=$1
mv $1 /tmp/$1
while [ $2 ]
do
        mv $2 $1
        shift
done
mv /tmp/$orig1 $1
```

set

- The `set` command sets the command line arguments
- It is useful for moving the output of command substitution into the command line arguments

```
$ date
Sat Feb 22 12:41:55 HKT 2014
$ cat setargs.sh
#!/bin/sh
set yat yih saam
echo "In Cantonese: 1 is $1, 2 is $2, 3 is $3"
set `date`
echo "Today is $3 $2 $6“
$ setargs.sh
In Cantonese: 1 is yat, 2 is yih, 3 is saam
Today is 27 Feb 2016
```

Special Parameters

Variable	Description
`$0`	The filename of the current script.
`$n`	The arguments with which a script was invoked. `n` is a positive decimal number corresponding to the position of an argument.
`$#`	The number of arguments supplied to a script.
`$*`	Stores all the arguments in a list of string
`$@`	Stores all the arguments as a single string
`$?`	Stores the exit status of last command. If last command runs successfully then it will be 0 and other value if not.
`$$`	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
`$!`	The process number of the last background command.

$$

- $$ is the process ID (PID) of the current process (the shell script PID, or the shell PID if interactive).

```
$ cat pid
#!/bin/sh
echo $$
$ pid
1154
$ pid
1156
$ pid
1157
$ echo $$
892
$ ps
PID TTY      TIME CMD
892 pts/0    0:01 tcsh
```

$$ (Cont.)

- It can be used for temporary file names:

```
$ cat swapfile2.sh
#!/bin/sh
file=/tmp/tmp$$
echo "Prepare a temp file name $file"
mv $1 $file
mv $2 $1
mv $file $2
$ swapfile2.sh
Prepare a temp file name /tmp/tmp5827
```

for Example: C-style for loop

```
$ cat for_randnum.sh
#!/bin/sh
for (( i=1; i<=5; i++ ))
        do echo "Random number $i: $RANDOM"
done

$ for_randnum.sh
Random number 1: 23320
Random number 2: 5070
Random number 3: 15202
Random number 4: 23861
Random number 5: 23435
```

for Example: keyword in

- Print out contents of all files under current directory

```
$ cat catall.sh
#!/bin/sh
     for file in *
     do
           if [ -f $file ]
           then
              echo "View $file [y/n]?"
              read resp
              if [ $resp = "y" ]
              then
                      cat $file
              fi
           fi
     done
```

for Example: Special Parameters

- If the “in ____” part is omitted, it defaults to $*

```
$ cat dollarstar.sh
#!/bin/sh
for file
do
      if [ -f $file ]
      then
              cat $file
      fi
done
$ dollarstar.sh file1 file2
This is file1
This is file2
```