COMP2113 Programming Technologies
ENGG1340 Computer Programming II

Module 2: Shell Script
Estimated time to complete: 2 hours

Objectives

At the end of this chapter, you should be able to:

- Understand what shell scripts are
- Write Bash shell script with a sequence of shell commands

Table of Contents

1. Shell Script

1.1 What is Shell Script

Shell script (.sh) is a computer program designed to be run by the Linux shell. It is an interpreted language, but not a compiled language. Unlike C++, we do not need to compile the shell into a binary executable format before executing the program. The program written in a shell script is parsed and interpreted by the shell every time the program is executed.

Interpreted languages allow us to modify the program more quickly by simply editing the script. However, the programs are usually slower because parsing and interpretation are needed during execution time.

Consider we have a sequence of shell commands, we don't want to re-type those commands whenever we want to execute. We can save them in a file and call that a shell script.

> Editing script file in Windows and importing to Linux may cause failure of execution because of the different end-of-line (EOL) used in Windows. You should create and edit script files inside a Linux system (e.g. use vi editor in SSH). Otherwise, you should ensure the line ending option in your text editor in Windows environment is set to UNIX format (LF) instead of Windows format (CRLF) if you import script files from Windows.

Here, we create a *hello.sh* script.

```
$ vi hello.sh
```

```
#!/bin/bash
echo "Hello world!"
echo "Welcome to ENEG1340"
```

*`echo` is a shell command that prints the string or value of a variable.

Make the script executable by the user:

```
$ chmod u+x hello.sh
```

Here is a sample run of the script:

```
$ ./hello.sh
Hello world!
Welcome to ENEG1340
```

All the scripts should start with **#!**. It indicates which program should be used to process the shell script. In this case, it is the path to the bash program. From Module 1 we know that there are many different shells (e.g., C shell, Korn shell, Bash shell). As we are using the Bash shell, we need to supply the path of the bash program so that the operating system knows how to interpret the bash shell commands. The path is **/bin/bash** in this case.

You can use the command `which -a` to locate the correct path to the Bash shell. The flag `-a` stands for returning all paths for the bash program.

```
$ which -a bash
/usr/local/bin/bash
/bin/bash
```

As you can see, the bash program maybe located in multiple locations. In the following, we will use the path **/bin/bash** for our shell scripts.

Another Example (echo -n)

Use flag `-n` if you do not want to output the trailing newline. For example *ex1_1.sh*:

```
#!/bin/bash
echo -n "Hello world!"
echo "Welcome to ENGG1340!"
echo "bye"
```

Sample run:

```
$ ./ex1_1.sh
Hello world!Welcome to ENGG1340!
bye
```

Because there is a **`-n`** flag with the first **echo** command, there is no trailing newline in the first output. The second output is then appended in the first line.

One more Example (with C++ program)

Write a script to compile, run and display result of a C++ program

add.cpp

```
//add.cpp
#include <iostream>
int main() {
    int a;
    int b;
    std::cin >> a;
    std::cin >> b;
    std::cout << a + b;
}
```

input.txt

```
3 4
```

ex1_2.sh

```
#!/bin/bash

#Compile the code
g++ add.cpp -o add
#Run the code and display result
./add < input.txt > output.txt
cat output.txt
```

Run *ex1_2.sh* in shell:

```
$ ./ex1_2.sh
7
```

2. Using Variables

There is only one variable type in shell scripts, which is **string**. Variable name is case sensitive. It can only contain letters (a - z, A - Z), number (0-9) or the underscore character (_).

2.1 Defining and Accessing Variables

This is how we can define a variable **pet** with initial value "**dog**".

```
pet="dog"
```

*No space is allowed before and after the = sign.

To access the value, use the dollar sign (**$**) with the variable name

```
#!/bin/bash

pet="dog"
echo $pet
```

The above script will give the following output.

```
dog
```

2.2 Read user input

The **read** command reads a string from the input and assigns it to the variable. For example, the following script read a user input and finally print it out.

```
1  #!/bin/bash
2
3  echo "What is your name?"
4  read name
5  echo "Hello $name"
```

- Line 4: **[Setting value]** We are setting the value of variable name (as the user input value). Therefore we do not need a dollar sign $ before name.
- Line 5: **[Retrieving value]** A dollar sign is required when retrieving the value of a variable. Therefore we have echo "Hello, $name" outputting "Hello, Kit"

Sample run:

```
$ ./ex2_2.sh
What is your name?
Kit
Hello, Kit
```

2.3 Quoting (single (') and double quote (""))

Quoting is very important on the command line and in shell script. There are 3 ways to specify a string value: **Unquoted**, **Single quote** and **Double quote**.

Unquoted

We can specify a string value without any quoting, but this method only works if the string value consists of a single word.

For example, the following will set the value of variable **a** as **cat**.

```
a=cat
```

However, it is wrong to create a string with a space in the value. With space, "`Apple`" is interpreted as a shell command. Therefore, the shell below will return "command not found".

```
a=Apple pie
```

Quoted

Any value between the pair of single/double quotation will be set as value of the string. However, single quotation does **not support variable substitution**.

With double quotation, a string can handles three special characters instead of directly including them into the strings.

Symbol	Description
$	**Dollar sign**: variable substitution.
\	**Backslash**: Escape special character
``	***Back quotes**: Enclose bash commands

* **Back quote button** is the button on the **left** of the button "1" in most keyboards.

For example, we declare a variable (name) and print the variable using **Single quote** and **Double quote**.

```
#!/bin/bash

name="Apple"
echo 'Hello, $name'
echo "Hello, $name"
echo "\$name = $name"
```

Sample run:

```
$ ./ex2_3.sh
Hello, $name
Hello, Apple
$name = Apple
```

In the example, the **$name** in the first echo command `'Hello, $name'` is NOT substituted by any variable as we use single quote.

For the second echo command `"Hello, $name"`, **$name** is substituted by the variable, so the output is "`Apple`" not "`$name`".

For the third echo command `"\$name = $name"`, **\$** is interpreted as a single dollar sign character so the output is "`$`" followed by "`name`". Same as above, **$name** is substituted by the variable (name).

2.4 Command Substitution

With backquotes (`), we can store the output of a shell command in a variable for further processing.

```
#!/bin/bash

a="`cat file.txt`"
echo $a

b="`wc -l file.txt | cut -d\" \" -f1`"
echo "There are $b lines in file"
```

file.txt

```
Apple
Banana
Cherry
```

Sample run:

```
$ ./ex2_4.sh
Apple Banana Cherry
There are 3 lines in file
```

In the above example, the command `wc -l file.txt` returns the result "**`3 file.txt`**", we then use the **`cut`** command to get the number of lines.

The cut command `cut -d" " -f1` is to separate the fields by the delimiter " " (i.e., a space) and return the first field (i.e., the output is "`3`"). However, the double quote in the command will match with the double quote at the beginning of the line (`b="`). To avoid this, we need to escape the double quote by adding a backslash (\) before it.

2.5 Operations on Strings

2.5.1 Get the length of a string

Getting the length of a string is very useful in shell script. Given a variable `a`, **`${#a}`** returns the number of characters in the value of variable `a`.

```
#!/bin/bash

a="Apple"
echo ${#a}
```

Sample run:

```
$ ./ex2_5_1.sh
5
```

`${#a}` returns the length of the string stored in variable **`a`**, which is 5.

2.5.2 Substring

Given any string variable `a`, **`${a:`*`pos`*`:`*`len`*`}`** returns the substring of `a` starting from position *pos* and has length *len*.

```
#!/bin/bash

a="Apple Pie"
echo ${a:6:3}
```

Sample run:

```
$ ./ex2_5_2.sh
Pie
```

Note: The index for the first character in a string is 0. Therefore `${a:6:3}` return the substring “Pie”.

Index	0	1	2	3	4	5	6	7	8
Character	A	p	p	l	e		P	i	e

3 chars

2.5.3 Replace a part of a string

Given any string variable `a`, **`${a/from/to}`** returns the string formed by replacing the **first** occurrence of *from* with *to*.

```
#!/bin/bash

a="Apple Pie"
from="Pie"
to="juice"
echo ${a/$from/$to}
```

Sample run:

```
$ ./ex2_5_3.sh
Apple juice
```

The first occurrence of "Pie" in "Apple pie" is replaced by "juice", therefore the output becomes "Apple juice".

2.5.4 Treat the value in a variable as number

It is less common to use shell scripts for mathematical calculations. Nevertheless, we can still perform mathematical operations using the **`let`** command. Normal operators like `+`, `-`, `*`, `/` and `%` are supported.

Example:

```
#!/bin/bash

a=10

let "b = $a + 9"
echo $b

let "c = $a * $a"
echo $c

let "d = $a % 9"
echo $d
```

Sample run

```
$ ./ex2_5_4.sh
19
100
1
```

2.6 Get command line argument in shell script

Command line arguments are labelled as `$0`, `$1`, ... `$9`. In particular, `$0` is the name of the shell script. Command line arguments after `$9` must be labelled as `${10}`, `${11}`, ..., as otherwise `$10` will be interpreted as `$1` and character `0`. The number of command line variables is given by `$#`.

For example, we have *ex2_6.sh* with the following content.

```
#!/bin/bash

echo "There are $# arguments"
echo "$0"
echo "$1"
echo "$2"
echo "$3"
echo "$4"
```

Sample run (with arguments):

```
./ex2_6.sh we are the world
There are 4 arguments
./ex2_6.sh
we
are
the
world
```

3. Flow of Control

3.1 If-else and condition

In this section, we will learn decision making in shell script.

The basic syntax of the if-statement is shown below.

```
if [ condition ]
then
    perform some action
fi
```

Be careful with the spaces between braces and expression. Missing these spaces will produce a syntax error.

The body of an if-statement is enclosed by the keywords "`then`" and "`fi`". "`elif`" is used as else-if.

```
if [ condition1 ]
then
    echo "condition 1 met"
elif [ condition2 ]
then
    echo "condition 2 met"
else
    echo "No condition met"
fi
```

Special syntax is needed to specify the condition in the if-statement. Listed below are the expressions that can be used in the condition.

1. String comparisons

   Notice that `$string1` and `$string` are enclosed with double quote so that comparison can work even if there are space inside `$string1` or `$string2`.

String comparisons	Description
`[ "$string" ]`	True iff the length of `$string` is non-zero
`[ "$string1" == "$string2" ]`	True iff the `strings` are equal
`[ "$string1" != "$string2" ]`	True iff the `strings` are different
`[ "$string1" \> "$string2" ]`	True iff `$string1` is sorted after `$string2`
`[ "$string1" \< "$string2" ]`	True iff `$string1` is sorted before `$string2`

2. File / Directory checking

   It is very convenient to do file checking in the conditions. We can simply provide the name of the file and check if the file exists, is a directory, etc.

File checking	Description
`[ -e $file ]`	True iff `file` exists
`[ -f $file ]`	True iff `file` is a file
`[ -d $file ]`	True iff `file` is a directory
`[ -s $file ]`	True iff `file` has size > 0
`[ -r $file ]`	True iff `file` is readable
`[ -w $file ]`	True iff `file` is writable
`[ -x $file ]`	True iff `file` is executable

3. Number comparison

   Although shell only has string variable, we can also perform number comparison by using the following expressions.

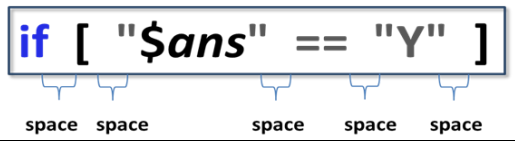
Number comparison	Description
`[ $a -eq $b ]`	True iff `a = b`
`[ $a -ne $b ]`	True iff `a != b`
`[ $a -lt $b ]`	True iff `a < b`
`[ $a -le $b ]`	True iff `a <= b`
`[ $a -gt $b ]`	True iff `a > b`
`[ $a -ge $b ]`	True iff `a >= b`

For example, this shell script asks for user approval to remove all .cpp files.

```
#!/bin/bash

echo "Do you want to remove all .cpp files? (Y/N)"
read ans
if [ "$ans" == "Y" ]
then
    rm -rf *.cpp
    echo "All .cpp files are removed!"
fi
```

Note that spacing is critical in shell script.



Sample run:

```
Do you want to remove all .cpp files? (Y/N)
Y
All .cpp files are removed!
```

3.1.1 Example (check file existence)

In this example, we write a shell script to check whether a file exists in the current directory.

```
#!/bin/bash

if [ -e hello.cpp ]
then
    echo "hello.cpp exist"
else
    echo "hello.cpp not found!"
fi
```

Suppose the file *hello.cpp* does not exist, the above script will produce the following result.

```
hello.cpp not found!
```

3.1.2 Example (check successful compilation)

Consider we use g++ to compile a .cpp file, how to know if the program is compiled successfully? If the compilation process is successful, then the executable will be generated.

Therefore, we can use the expression `[ -e file]` to check if the compilation is successful.

```
#!/bin/bash

if [ -e hello.cpp ]
then
    echo "hello.cpp exist"
    g++ hello.cpp -o hello
    if [ -e hello ]
    then
        ./hello
    fi
else
    echo "hello.cpp not found!"
fi
```

Suppose we have the file *hello.cpp* and it is compiled successfully, the script will produce the following result

```
hello.cpp exist
Hello World!
```

3.1.3 Example (handle compilation error message)

Consider we use g++ to compile a .cpp file, if the compilation process is not successful, then the executable will NOT be generated. Therefore, we can use [-e *file*] to check its existence. How about viewing the error message outputted by the g++ compiler?

We can use file redirection technique to redirect error message to a file.

```
g++ hello.cpp -o hello 2> error.txt
```

For example:

```
#!/bin/bash

if [ -e hello.cpp ]
then
    echo "hello.cpp exist"
    g++ hello.cpp -o hello 2> error.txt
    if [ -e hello ]
    then
        ./hello
    else
        echo "Compilation failed!"
        echo "Here are the error message"
        cat error.txt
    fi
else
    echo "hello.cpp not found!"
fi
```

Suppose *hello.cpp* contains a syntax error; the following will be outputted.

```
hello.cpp exist
Compilation failed!
Here are the error message
hello.cpp: In function ‘int main()’:
hello.cpp:5:5: error: expected ‘;’ before ‘return’
     return 0;
     ^~~~~~
```

3.2 For-loop

With loop, we can execute a set of commands repeatedly. The for-loop operates with a list of items. The body of the loop is enclosed with keyword `do` and `done`.

Here is an example of using a for-loop to print 1 to 5.

```
#!/bin/bash

list="1 2 3 4 5"
for i in $list
do
    echo "This is iteration $i"
done
```

The value of variable **`$list`** is spitted by space. Therefore, there are 5 items. Each item becomes the value of the variable `i` in each iteration. **(The technique to split the values by other delimiters is out of syllabus.)**

Upon execution, you will receive the following output.

```
This is iteration 1
This is iteration 2
This is iteration 3
This is iteration 4
This is iteration 5
```

3.2.1 Example (Loop through files)

Other than pre-setting a list, you can also loop through files in a directory.

For example, you can write a shell script to backup all .cpp files in the current directory.

```
1  #!/bin/bash
2
3  list="`ls *.cpp`"
4  for fileName in $list
5  do
6      cp $fileName "$fileName.backup"
7  done
```

Sample run:

```
$ ls
a.cpp  backup.sh*  b.cpp  c.cpp
```

```
$ ./backup.sh
```

```
$ ls
a.cpp  a.cpp.backup  backup.sh*  b.cpp  b.cpp.backup  c.cpp  c.cpp.backup
```

Explanations

Line 3:

```
list="`ls *.cpp`"
```

With backquote, `ls *.cpp` is regarded as a shell command, and the result of the command (i.e., a.cpp b.cpp c.cpp) is stored in the variable `$list`.

Line 4:

```
for fileName in $list
```

With for loop, the variable `fileName` represents one item in the `$list` in each iteration, therefore $fileName=a.cpp in the first iteration, $fileName=b.cpp in the second iteration and $fileName=c.cpp in the last iteration.

Line 6:

```
cp $fileName "$fileName.backup"
```

The **`cp`** command copies the first argument to the second argument. In the 2nd argument, we use variable substitution to append the original file name with **.backup**.

4. Useful techniques in shell script

4.1 Hide unwanted command output in shell script

Shell commands generate its own error and output message, which may mess up the output of your shell script.

To get rid of the system generated errors and outputs, simply use file redirection technique (`1>` and `2>`) on the shell commands.

```
#!/bin/bash

cp file123 fileabc 1>/dev/null 2>&1
if [ -e fileabc ]
then
    echo "Copy successful"
else
    echo "$0: Oops. Copy failed :("
fi
```

`1>/dev/null` redirects the standard output (denoted by `1>`) of the `cp` command to the system dustbin */dev/null*, somewhere we can redirect the output of a command to.

`2>&1` redirects the standard error (denoted by `2>`) of the `cp` command to the same location as where we redirect the standard output (denoted by `&1`).

Sample run:

```
$ ./ex4_1.sh
./ex4_1.sh: Oops. Copy failed :(
```

4.2 Output to standard error

The shell script itself can also `echo` content to standard error.

```
#!/bin/bash

cp file123 fileabc 1>/dev/null 2>&1
if [ -e fileabc ]
then
    echo "Copy successful"
else
    echo "$0: Oops. Copy failed :(" >&2
fi
```

With `>&2` at the end of the echo command, we are outputting the echo message to the standard error, useful when we are outputting error messages in the shell script!

Therefore, if we run it and redirect standard error to *error.txt* using the following command:

```
$ ./ex4_2.sh 2> error.txt
```

The error message will not be output on screen but directed to *error.txt* instead.

```
$ cat error.txt
./ex2_6.sh: Oops. Copy failed :(
```

Now your shell script performs like other standard shell commands.

5. Further reading

We have not covered topics like while-loop and functions (they are out of syllabus). Students interested in learning more about shell script programming techniques can refer to the following references:

- Chapter 18. Control flow structure in Linux & UNIX shell programming, David Tansley, Addison-Wesley.
- Advanced Bash-Scripting Guide. An in-depth exploration of the art of shell scripting http://tldp.org/LDP/abs/html/index.html
- Linux Shell Scripting Tutorial (A Beginner's handbook) http://www.freeos.com/guides/lsst/

6. References

- Chapter 16. Introduction to shell script in Linux & UNIX shell programming, David Tansley, Addison-Wesley.
- Chapter 11.1 The interactive bash shell in UNIX shells by example, 3rd / 4th edition, Ellie Quigley, Prentice Hall.
- Getting Started in Programming by John S. Riley. Chapter 4.1. Interpreted vs. Compiled Languages http://www.dsbscience.com/freepubs/start_programming/start_programming.html
- Wikipedia Shebang (Unix) http://en.wikipedia.org/wiki/Shebang_(Unix)
- Chapter 5. Shell input and output and Chapter 14 Environment and shell variables in *Linux & UNIX shell programming*, David Tansley, Addison-Wesley.
- Chapter 11. The interactive bash shell and Chapter 12 Programming with the bash shell in *UNIX shells by example*, 3rd / 4th edition, Ellie Quigley, Prentice Hall.
- Bash Reference Manual http://www.gnu.org/software/bash/manual/html_node/index.html#SEC_Contents
- Article: What's the Difference Between Single and Double Quotes in the Bash Shell?
- http://www.howtogeek.com/howto/29980/whats-the-difference-between-single-and-double-quotes-in-the-bash-shell/
- Chapter 18. Control flow structure in Linux & UNIX shell programming, David Tansley, Addison-Wesley.
- Chapter 12 Programming with the bash shell in UNIX shells by example, 3rd / 4th edition, Ellie Quigley, Prentice Hall.