Faculty of Computer Science, Dalhousie University *28-Nov-2018*

CSCI 2132 — Software Development

Lecture 33: Shell Scripting

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

Previous Lecture

- File Manipulation in C:
- Opening a file, closing a file
- Formatted I/O, character I/O
- Block reading and writing
- File positioning
- Example with file writing

26 Shell Scripting

The Unix shells provide features for programming, similar to programming languages. This is called shell programming, or shell scripting. We have spent a lot of time learning C, which is already closely related to Unix. Why spend time learning shell scripting? There are a few reasons. First, shell scripting is very useful for system administration. When a shell starts, some scripts are run automatically to configure the environment. Shell scripting is also useful for fast prototyping, as it is easy to use many Unix utilities in shell scripts, and no compiling is required before we run a shell script. Finally, this reflects the Unix philosophy of breaking projects into sub-tasks, as it allows us to combine a set of Unix utilities to perform a task.

26.1 Shell Programs (Scripts)

A shell program is a text file that stores a series of shell commands. After we grant it the execute permission, we can run this script. Let us see a very simple shell script that contains a few simple Unix commands. We name this script `current.sh` (it is not necessary to use .sh as the suffix of a script).

```
#!/bin/bash
#Print current status
whoami
pwd
ls
```

In this shell script, the first line tells us it is a script for the Bash shell. The second line is a comment; `#` functions as `//` in C99 or Java; i.e., it is a line comment. This script, when it runs, executes the three commands that are given in lines 3–5. To run this, we need use `chmod` to grant the user execute permission to it. For example, if we would like to let the owner of the script execute it, we can enter:

```
chmod u+x current.sh
```

After that, we can run the script by entering its pathname:

```
./current.sh
```

Let us learn more rules of specifying which shell this program is for using the first line of the script. If the first line contains only a #, then the shell from which the script is executed is used. If it is of the format `#!pathname`, then it is for the shell whose executable program's pathname is the one given in the line. This is what we used in the example above. If neither rule applies, then the script is for the Bourne shell. This first line is necessary since there are many Unix shells, and their shell programming languages have different syntax and features. It is preferable to specify the pathname of the shell explicitly. This rule is not only used for the Shell programs, but for any other program that can interprent a program like this. For example, a similar line is used for Perl and Python programming langauges.

26.2 Shell Variables

We can use variables in shell programs. Previously we learned how to use variables in the shell, and the same rules apply to variables in shells scripts. To create a variable, we simply assign a value to a variable using =. To access its value, we use `$`. The following is an example that we can use in a shell script:

```
i=1
echo $i
```

Variable names can contain letters, digits and underscores, but they cannot start with digits. When we use =, we cannot use any space characters before and after it.

There are some predefined local variables that we can directly use in a shell script. The following predefined local variables are related to command-line arguments:

- `$0` is the pathname of the script
- `$n` is the n-th command arguments. We can use `$1`, `$2`, ..., `$9`, `${10}`, `${11}`, ...
- `$#`: the number of command-line arguments, excluding `$0`

26.3 Arithmetic Operations

To use an arithmetic expression in Bash shell script statement, we place it inside double parentheses:

```
(( expressions ))
```

By doing so, we can use spaces when needed to make the expression look better.

These arithmetic operators are provided: =, +, −, ++, −−, *, /, %, and **. All but ** have the same meaning as the corresponding operators in C. The operator ** is the exponentiation operator. For example, `2**3` means 2 to the power of 3.

We can also use `()` to change precedence.

The following example is a script named `add.sh`, which adds the two command-line arguments that the user inputs as integers, and prints the result:

```
#!/bin/bash
(( sum = $1 + $2 ))
echo the sum of $1 and $2 is $sum
```

There are some common mistakes. The following two lines are examples:

```
(( sum = 1 + 2 ))

(( $sum = $1 + $2 ))
```

26.4 Conditional Expressions

Conditional expressions are used for control structures. One type of conditional expressions is for arithmetic test. The syntax is the same as that for arithmetic expressions:

```
(( expressions ))
```

The following operators can be used in arithmetic test: <=, >=, <, >, ==, !=, !, &&, and ||. There are operators for comparing strings. To write an expression that compares strings, the syntax is

```
[ expression ]
```

There must be a space after [and before]. We can use operators == and != to compare strings. Two additional string operators are: -n *string*, and -z *string* (nonzero and zero length).

26.5 Control Structures

There are if statements in shell programming. They function the same way as the if statements in C, but the syntax is quite different:

```
if condition1; then
  commands
elif condition2; then
  commands
else
  commands
fi
```

The elif and else parts are optional. The following example rewrites the add.sh script, so that we can print an error message if the user does not enter two integers as command-line arguments when running the script:

```
#!/bin/bash

if (( $# != 2 )); then
  echo usage: ./add.sh num1 num2
  exit
fi

(( sum = $1 + $2 ))
echo the sum of $1 and $2 is $sum
```

In the above example, the command exit is used to terminate a shell program.

Example with Arithmetic for-Loop

```
#!/bin/bash
```

```
if (( $# != 1 )); then
    echo usage: $0 num1
    exit
fi

for (( i = 1; $i <= $1; i = $i + 1 )) do
  f=tmpfile-$i.txt
  echo "Appending file $f"
  echo Updated on `date` >> $f
done
```

'Standard' Bash For Loop

However, a more standard 'for' loop in Bash is the following form of the for loop.

The syntax of this loop is significantly different from C or Java 'for' statements. The following is the syntax:

```
for var in word {word}*
do
  commands
done
```

Here word `{words}*` is a list of words separated by white-space characters. In each iteration, the variable `var` takes one word as its value. This may seem weird, but it is useful, as it allows us to use the output of Unix commands in the for loop. Let us take advantage of this to write a program that sorts the contents of all `.txt` file in a directory, and store them in `*.txt.sorted` files:

```
#!/bin/bash
for file in *.txt
do
  sort $file > $file.sorted
done
```

The Bash shell requies new-lines after the commands, but instead we can use the semi-colon (`;`). Hence the above example can also be written as:

```
#!/bin/bash
for file in *.txt; do
  sort $file > $file.sorted
done
```

or

```
#!/bin/bash
for file in *.txt; do sort $file > $file.sorted; done
```

Alternatively, we can use a feature of the Bash shell to perform the same task. This feature is called command substitution. If we place a command within a pair of back-quotes (`) and then include it as part of a command line, then this command will be executed, and its output will be inserted in the command line. For example, the following command line can be used to print a text message that tells us how many files there are in the current working directory:

```
echo There are `ls | wc -l` files in the current directory
```

Thus, the following shell script can perform the same task that the last example performs:

```
#!/bin/bash
for file in `ls *.txt`
do
  sort $file > $file.sorted
done
```

Case Statement

- Similar to the switch statement in C or Java; syntax:

```
case var in
  word{|word}*)
    commands
    ;;
  ...
esac
```