UT 012.
INTRODUCTION TO SHELL SCRIPTING

Computer Systems
CFGS DAW / DAM

Álvaro Maceda
a.macedaarranz@edu.gva.es
2022/2023
Version:240321.1202

License

Nomenclature

Throughout this unit different symbols will be used to distinguish important elements within the content. These symbols are:

 Important

 Attention

 Interesting

TABLE OF CONTENTS

UT 012. Introduction to shell scripting

1. Control structures

1.1 Loops

Loops are another basic programming structure. In Bash we can use `for` and `while` loops to implement this structure.

1.1.1 For loop

`for` loops iterate through a list of values. For example:

```
for i in 1 2 3 4 5
do
  echo "Looping ... number $i"
done

Looping ... number 1
Looping ... number 2
Looping ... number 3
Looping ... number 4
Looping ... number 5
```

You can use glob patterns for the `for` loop. The glob pattern will be replaced by the files matching that pattern, but if globbing fails to match any files/directories, the original globbing character will be preserved:

```
for item in asereje file* sarandonga
do
  echo "The item is $item"
done

The item is asereje
The item is file00.txt
The item is file01.txt
The item is sarandonga
```

It's also possible to use command substitution in `for` loops:

```
counter=0
for file in $(ls -1)
do
  counter=$((counter+1))
  echo "File $counter is $file"
done

File 1 is ...
File 2 is ...
File 3 is ...
```

Internal field separator (IFS)

If you use a string as a parameter for the for loop, the string will be split in "words", and each word will be taken as a different input for the loop:

```
data="This will run five times"
for item in $data
do
  echo "The item is $item"
done

The item is This
The item is will
The item is run
The item is five
The item is times
```

The behaviour is the same if we use the string directly. This happens because the strings are split using a special character, `IFS`. `IFS` stands for "Internal Field Separator" and is a special variable used in shell programming to specify the delimiter that separates fields in a string.

By default, the `IFS` variable is set to whitespace characters (i.e., space, tab, and newline), which means that a string will be split into fields whenever any of these characters are encountered. However, you can change the value of `IFS` to specify a different delimiter character or set of characters. For example:

```
IFS=":|"
data="different:fields|one:variable"
for item in $data; do
  echo "The item is $item"
done

The item is different
The item is fields
The item is one
The item is variable
```

C-like loop

There is a special syntax for running for loops with counters similar to common programming languages:

```
for ((i = 0 ; i < 3 ; i++)); do
  echo $i
done

0
1
2
```

Ranges

Using the Bash Sequence Expression function `{START..END..INCREMENT}` we can generate ranges of integers or characters, defining a start and an end. This function is often used in conjunction with for loops:

```
for i in {1..5..2}
do
  echo "Number: $i"
done

Number: 1
Number: 3
Number: 5
```

1.1.2 While loop

In Bash scripting, a `while` loop is a control flow statement that allows you to execute a block of code repeatedly while a certain condition is true. The general syntax of a while loop in Bash is as follows:

```
while [ condition ] # You can also use double-bracket syntax here
do
  # block of code to be executed
done
```

Let's consider an example. The following script prints out the numbers from 1 to 5 using a while loop:

```
n=1
while [ $n -le 5 ]; do
  echo $n
  n=$((n+1))
done
```

Infinite loops

You can also create infinite loops in bash using the while syntax with a condition that evaluates always to true. You can do that with `true` and `:`. In case you create an infinite loop, you will need one of the exit, break or continue commands to exit that loop. For example:

```
while : # You can also use while true
do
  read -p "Enter a value: " input
  if [ "$input" = "exit" ]; then
    break
  fi
  echo "You entered: $input"
done
```

The program will print the values until the user enters `exit`.

1.1.3 Processing files with loops

In Bash, you can read files with loops using various commands such as `while` and `for`. With `while` loops, you can use redirection in combination with the `read` command to process a file line by line:

```
filename='whatever.txt'
while read $LINE; do
  echo "The line is: $LINE"
done < $filename
```

You can also redirect the output of a command to a while loop:

```
cat file.txt | while read line; do
  echo "$line"
done
```

Similarly, you can use the for loop to read a file using subcommands:

```
for line in $(cat /path/to/file.txt); do
  echo "$line"
done
```

1.2 Exit, break and continue

In shell scripting, exit, break, and continue are control flow commands that allow you to modify the behaviour of loops and scripts.

The `exit` command is used to exit a shell. This command terminates the current Bash shell session and returns the user to the parent shell or operating system prompt. It can be used with a numeric argument that represents an exit status code. The exit status code is used to indicate the success or failure of the command or script that was executed. By convention, a value of `0` represents success, and any non-zero value represents an error or failure.

```
if [ number -lt 0 ]; then
  exit 1 # Exit with error status code
fi
echo "The number is >= 0, can proceed"
```

`break` is used to exit out of a loop. When break is called inside a loop, the loop terminates immediately and control is transferred to the next statement after the loop. This is useful for situations where you want to exit a loop early based on some condition:

```
while true; do
  read -p "Guess the number:" NUMBER
  if [ $NUMBER -eq $GUESS ]; then
    echo "You found the number"
    break
  fi
done
```

The `continue` command is used to skip the current iteration of a loop and move on to the next iteration. This is useful for situations where you want to skip certain iterations of a loop based on some condition.

```
echo "Printing only files in the current directory:"
for file in *; do
  if [[ ! -f $file ]]; then
    continue
  fi
  echo $file
done
```

2. Arguments

So far, everything we have put in a script could be done directly from the console. However, when we work with scripts we can use something additional, which is the script arguments. In Bash scripts, arguments are values passed to the script when it is executed. These arguments allow you to customize the behaviour of the script and make it more flexible.

To pass arguments to a Bash script, you can specify them after the script name, separated by spaces. For example, if you have a script named `my_script.sh` and you want to pass two arguments to it, you would run it like this:

```
./my_script.sh -foo argument2
```

In this example, `-foo` and `argument2` are the two arguments being passed to the script.

Within the Bash script, you can access these arguments using special variables, starting with $1 for the first argument, $2 for the second argument, and so on. For example, to print the first argument, you can use the echo command like this:

```
echo "The first argument is: $1"
echo "The second argument is: $2"
```

In the above example, the script will print:

```
The first argument is: -foo
The second argument is: argument2
```

You can also use the special variable `$#` to get the total number of arguments passed to the script, and `$0` to get the name of the script itself.

3. Supplementary material

Loops in Bash

https://ryanstutorials.net/bash-scripting-tutorial/bash-loops.php

https://linuxhandbook.com/bash-loops/

Reading files with loops

https://www.shellscript.sh/loops.html

Parameters

https://tecadmin.net/tutorial/bash-scripting/bash-command-arguments/