BUSA8090 - Assignment 1, Task 1

Samuel Gardiner (44952619)

17 March 2020

Note to the marker:

To easily download and make executable all of the scripts contained in this assignment, please run the following command in your shell:

```
git clone https://github.com/gardiners/a1t1 && chmod u+x a1t1/*.sh
```

Question 1

We present a script `newer.sh` which, when given a list of filenames as its command-line arguments, prints the name of the newest file. Executable source code is available at `https://raw.githubusercontent.com/gardiners/a1t1/master/newer.sh`

```
1  #!/bin/bash
2
3  # newer: given a list of filenames as arguments, returns the name of the
4  # newest file.
5
6  # If the arguments are empty, print usage help text and quit.
7  if [ -z "$*" ]; then
8          echo "Usage: ./newer.sh [FILE]..."
9          exit 1
10 fi
11
12 # Sort the argument filenames by time, and store as a bash array.
13 sorted=($(ls -t $*))
14 # Return the first element in the array (ie the newest file).
15 echo ${sorted[0]}
```

The `if` conditional at lines 7-10 checks whether the user has given any arguments. If the list of arguments (that is, `$*`) is empty, our script prints a helpful usage message, and then quits.

The heavy lifting is performed by the command `ls -t $*` on line 13. The `-t` argument tells `ls` to sort its output by file time, and we have provided our list of script-calling arguments `$*` as the input. The output of `ls` is captured by the `$()` construct, and then captured again by an outer pair of parentheses to form a `bash` array which we have named `sorted`.

To print the name of the newest file, we simply return the first element (ie element `[0]`) of our array at line 15.

To test whether the script works, we create three files with known modification times and check whether `newer.sh` correctly returns the newest:

```
ubuntu@ip-172-31-20-200:~/busa/a1t1$ touch -t 202003151800 foo
ubuntu@ip-172-31-20-200:~/busa/a1t1$ touch -t 202003151801 goo
ubuntu@ip-172-31-20-200:~/busa/a1t1$ touch -t 202003151802 hoo
ubuntu@ip-172-31-20-200:~/busa/a1t1$ ./newer.sh foo goo hoo
hoo
```

As expected, the script returns the newest file, `hoo`, which has a modification time a minute later than `goo` and two minutes later than `foo`. This remains the case if we change the order of the filename arguments:

```
ubuntu@ip-172-31-20-200:~/busa/a1t1$ ./newer.sh goo hoo foo
hoo
```

If we specify a filename that doesn't exist, we get a useful error message from `ls` itself on `stderr`, but still get the newest of the files that we correctly specified:

```
ubuntu@ip-172-31-20-200:~/busa/a1t1$ ./newer.sh foo bar hoo goo baz
ls: cannot access 'bar': No such file or directory
ls: cannot access 'baz': No such file or directory
hoo
```

Finally, if we specify no filenames at all, we get a helpful message explaining how to use the script:

```
ubuntu@ip-172-31-20-200:~/busa/a1t1$ ./newer.sh
Usage: ./newer.sh [FILE]...
```

Question 2

We present our script `test_me.sh`, which prints the text "This is a TEST" to the terminal if called with any arguments, but prints "This is NOT a test" if called without an argument. Source code is available at `https://raw.githubusercontent.com/gardiners/a1t1/master/test_me.sh`

```
1  #!/bin/bash
2
3  # test_me: echoes one string if given no arguments, and another string if given
4  # any arguments
5
6  # Is the number of arguments zero?
7  if ! (($#)); then
8          # Then we have no arguments:
9          echo "This is NOT a test"
10 else
```

```
11          # Therefore we have at least one argument:
12          echo "This is a TEST"
13  fi
```

The conditional `if` at line 7 checks the number of arguments. Since the integer 0 is interpreted as Boolean FALSE within `bash`'s `((expression))` arithmetic expansion syntax, if we have no arguments, `! (($#))` will evaluate to TRUE. In this case, the '`then`' branch of the conditional executes and we get the result "This is NOT a test". The complement of `! (($#))` is obviously `(($#))` (the case where we have any arguments) and in this circumstance the '`else`' branch is executed instead, yielding the message "This is a TEST".

We can test our script under either of these conditions:

```
ubuntu@ip-172-31-20-200:~/busa/a1t1$ ./test_me.sh
This is NOT a test
ubuntu@ip-172-31-20-200:~/busa/a1t1$ ./test_me.sh foo
This is a TEST
ubuntu@ip-172-31-20-200:~/busa/a1t1$ ./test_me.sh foo bar baz
This is a TEST
```

We can see that our script has returned the desired output in the zero-argument case, the one-argument case, and the many-argument case.

Question 3

a)

Program 24 is the shell script `time-signal.sh`, printed at Wünschiers 10.11.2. The script is provided online by Wünschiers with the URL `https://www.staff.hs-mittweida.de/~wuenschi/data/media/compbiolbook/chapter-10-shell-programming--time-signal.sh`. Since this is a publicly available URL, we can easily use `curl` to read the script from Wünschiers' webserver and write it to a directory on our Ubuntu instance. `>`.

First, we create the directory `~/bin`:

```
ubuntu@ip-172-31-20-200:~$ mkdir -p ~/bin
ubuntu@ip-172-31-20-200:~$ ls -lah ~/bin
total 8.0K
drwxrwxr-x  2 ubuntu ubuntu 4.0K Mar 17 10:23 .
drwxr-xr-x 12 ubuntu ubuntu 4.0K Mar 17 09:04 ..
```

We have used the `-p` argument to `mkdir` as it prevents `mkdir` from generating an error if the directory already exists (for example, in the case that this command is run by a peer marker).

Now, we can write the file `~/bin/time-signal.sh` using `curl`, and set it to be executable with `chmod`:

```
ubuntu@ip-172-31-20-200:~$ curl -o ~/bin/time-signal.sh
↪  https://www.staff.hs-mittweida.de/~wuenschi/data/media/compbiolbook/
   chapter-10-shell-programming--time-signal.sh
```

```
 % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   224  100   224    0     0    127      0  0:00:01  0:00:01 --:--:--   127
ubuntu@ip-172-31-20-200:~$ chmod u+x ~/bin/time-signal.sh
ubuntu@ip-172-31-20-200:~$ ls -lah ~/bin/time-signal.sh
-rwxrw-r-- 1 ubuntu ubuntu 224 Mar 17 10:42 /home/ubuntu/bin/time-signal.sh
```

We elected to use the `-o` (output file) switch for `curl` to specify the destination, although we also could have used the file redirect operator `>`.

b)

From `man bash`:

```
((expression))
      The expression is evaluated according to the rules described below under
      ↪  ARITHMETIC EVALUATION.  If the value of the expression is non-zero,
      ↪  the return status is 0; otherwise the return  status is 1.  This is
      ↪  exactly equivalent to let "expression".
...
ARITHMETIC EVALUATION
       The shell allows arithmetic expressions to be evaluated,  under
       certain  circumstances  (see  the  let and declare builtin com-
       mands, the ((  compound  command,  and  Arithmetic  Expansion).
...
       = *= /= %= += -= <<= >>= &= ^= |=
              assignment
...
        Within an expression, shell variables may also be referenced by name
        without using  the parameter  expansion  syntax.
```

Which we compare to the current form of line 9 of `time-signal.sh`:

```
count=$[$count+1]
```

So, the syntax `let count=count+1` evaluates `count+1` and reassigns it to `count`, which is the same outcome as performing the evaluation using arithmetic expansion. When using `let`, the `$` can be omitted from the variable names.

c)

The `expr` command evaluates arithmetic expressions given as its arguments. The `$()` construct captures its output and it is assigned to the variable `count`. Examining the entire command

```
count=$(expr $count + 1)
```

We can therefore construct the sequence of operations that yields the result:

- `$count` is substituted with its value (hopefully an integer)
- `expr` evaluates its arguments

- `$()` captures the results of executing `expr`; and
- the result is reassigned to `count`.

This is the same outcome as the original code (that is, incrementing the value of `count`).

d)

From `man bash`:

```
Arithmetic Expansion
    Arithmetic expansion allows the evaluation of an arithmetic expression and
    ↪ the substitution of the result.  The format for arithmetic expansion
    ↪ is:

        $((expression))

    The old format $[expression] is deprecated and will be removed in upcoming
    ↪ versions of bash.
```

So, the expressions `((count+1))` and `$[$count+1]` are functionally identical, although the latter form should be avoided as it may stop working in a future version of `bash`. The inner `$` symbol could safely be omitted in the original code, in keeping with `bash`'s rules for arithmetic expansion and arithmetic evaluation.

Both versions of the code increment `count`.

e)

We have altered the `time-signal.sh` script to meet the question requirements, and present it below. Source code is available at `https://raw.githubusercontent.com/gardiners/a1t1/master/time-signal.sh`

```
#!/bin/bash
# Modified time-signal.sh with more chiming.

# Extract the chiming loop into a function for reuse throughout the script.
# Takes one argument: the number of chimes.
function chime {
        for ((i = 0; i < $1; i++ )); do
                echo -e "\a"
                sleep 1
        done
}

# Store the hour and minute:
hours=$(date +%I)
minutes=$(date +%M)

# Chime the hours and then wait three seconds:
chime $hours
```

```
19  sleep 3
20  # Chime the minutes depending on which quarter of the hour we are in:
21  chime $((minutes / 15))
```

At lines 6-11, we define a `bash` function called `chime` which takes a single argument, the number of chimes to loop through. We have done this rather than repeat the chime loop code at multiple points in our script. Within the function definition, `$1` evaluates to the function's first argument, not the script's first argument. We have used a slightly more concise and C-like `for` loop syntax than Wünschiers' original script. This is a stylistic choice only and does not affect the operation of the loop in any way.

At lines 14 and 15 we store the hour and minute values for the present time. The datetime formatting codes can be found in `man date`:

```
    %I      hour (01..12)
    %M      minute (00..59)
```

We then chime the hours, as in the original script, at line 18. The three-second pause is executed at line 19.

Chiming the number of quarter hours (line 21) relies on the fact that `bash` arithmetic evaluation with `$((expression))` is performed on integers only. So, the division operator `/` actually performs integer division, and returns only the integer quotient. Therefore, at m minutes past the hour,

- when $0 \leq m < 15$, `$((minutes / 15))` $= 0$
- when $15 \leq m < 30$, `$((minutes / 15))` $= 1$
- when $30 \leq m < 45$, `$((minutes / 15))` $= 2$
- when $45 \leq m < 60$, `$((minutes / 15))` $= 3$

Another approach to solving this problem would have been using a `case ... esac` construct with a series of expressions to test which part of the hour `$minutes` falls within, but the approach in our script is more concise and arguably more elegant.