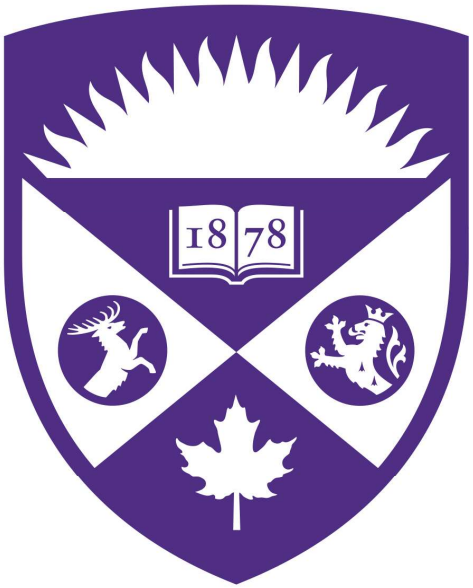


Western

UNIVERSITY · CANADA

Shell programming

Winter 2022

Shell programming

- Bash can accept a file as an argument
- This file can have a list of commands to run under the Bash shell (usually one per line)
- Bash understands variables, loops, and conditional logic
- Therefore, we can write programs for Bash (we call this "scripting" since there is no compilation step)

Shell programming

- We typically write Bash scripts to automate complex and/or repeatable tasks
- To run a Bash script, any of the following will do:
  - `bash <scriptname>`
  - `./<scriptname>` - assuming the script is executable. Use `chmod` to make it executable

"Shebang"

- A script will use whatever shell invoked it
- It is typically good practice to manually specify what shell your script expects
- This is accomplished with the "hash-bang" line (`#!<shell>`) which appears on the first line. E.g.
  - `#!/bin/bash`
  - `#!/bin/zsh`

"Shebang"

- Since `/bin/sh` is installed on any Unix system and the difference between `/bin/sh` and `/bin/bash` for scripting is negligible, it is more common to use `#!/bin/sh`

- Use
  `#!/bin/sh -x`
  to force your script to print out every line that gets executed before it is executed. This is useful for debugging

Comments

- Bash uses single-line comments only (like // in C)
- Use the # character to insert a comment

```
echo "my output" # This is a comment
```

Variables

- Create a variable by assigning a value to a string
  - `myvar=myvalue`
- Access the variable by using the ${variablename}
  - `${myvar}`
- `$myvariablename` without the curly braces is also okay but it is less safe

```
myvar1="String value"
myvar2=5    #This is an integer
echo "myvar1=${myvar1}, myvar2=${myvar2}"
```

Control statements

- Bash supports three types
  - Conditionals – if/elif/else, case
  - Loops – for, while, until, do-while
  - Branches – subroutines and gotos

Control statements

- In Bash, we can only test the return value of a command
- Any command can be run and we check the return value to decide if the statement is true or false
- Bash is somewhat counter intuitive
  - 0 – True (ie. No change in state – status quo)
  - Non-zero – False (ie. Something is different)
- Check the exit codes in the man pages

Control statements

- The latest exit code is always contained in the `$?` variable

```
[wbeldman@compute bin]$ echo "SUCCESS"
SUCCESS
[wbeldman@compute bin]$ echo $?
0
[wbeldman@compute bin]$ echo \
> ^C
[wbeldman@compute bin]$ echo $?
130
[wbeldman@compute bin]$
```

Control statements

- We use the test command to test our variables and check it's return value
  - `test expression`
- Bash uses [ ] as a shorthand for the test command
  - `[ expression ]`

Control statements

- Bash extends the single square brackets to double square brackets to allow more common comparisons with a more natural syntax (e.g. ==, !=, <, >)
  - `[[ expression ]]`

Control statements

- To see the entire list of tests, see the man page for test
  - There are tests for
    - Files and directories
    - Strings
    - Integers

If/elif/else statements

```
myvar=-2

if [ ${myvar} -gt 0 ]; then
    echo "${myvar} is positive and > 0"
elif grep ${myvar} somefile.txt 2>/dev/null; then
    echo "This is an example with the grep command"
else
    echo "${myvar} is negative and isn't found in somefile.txt"
fi
```

For loops

```
#!/bin/sh
# timestable - print out a multiplication table
for i in 1 2 3; do
    for j in 1 2 3; do
        value=`expr $i \* $j`
        echo -n "$value "
    done
    echo
done
```

For loops

```
#!/bin/sh
# timestable - print out a multiplication table
# ((»   )) is a compound expression
#           Evaluate each sub-expression arithmetically
for ((i=1; i<4; i++)); do
    for ((j=1; j<4; j++)); do
        (( value = i*j ))
        echo -n "$value "
    done
    echo
done
```

For loops

```
#!/bin/sh
# file-poke - tell us stuff about files
files=`ls`
for i in $files; do
    echo "$i"
    stat --printf="\t%U %G %A" $i
    echo
    echo
done
```

While loops

```
#!/bin/sh
# timestable - print out a multiplication table
# ((»   )) is a compound expression
#           Evaluate each sub-expression arithmetically
i=1
while (( i < 4 )); do
    j=1
    while (( j < 4 )); do
        (( value = i*j ))
        echo -n "$value "
        ((j++))
    done
    echo
    ((i++))
done
```

While loops

```
#!/bin/sh
i=1
sum=0
while [ $i -le 100 ]; do
    sum=`expr $sum + $i`
    i=`expr $i + 1`
done
echo The sum is $sum.
```

Until loops

```
#!/bin/sh
x=1
until [ $x -gt 3 ]; do
    echo x = $x
    x=`expr $x + 1`
done
```

Semi-colon

- It is safest to end every command in a script with a semi-colon
- Semi-colon also allows multiple commands on one line

```
[wbeldman@compute bin]$ echo "HERE ARE MY FILES"; ls;
HERE ARE MY FILES
decrypt.sh  encrypt.sh  test.sh
```

Arguments

- Our script can accept arguments (just like argv/argc in C)
- We access these arguments through the `$N` variable where N is the argument position
- `$#` is an integer representing the number of arguments
- `$@` and `$*` is a list of all arguments

Prompting for input

- Use the read command to load user input into variables
- E.g. To display a prompt and read 3 variables from the user, where each variable is separated by a space:
  - ```
    read -p "  This is my prompt: " var1 var2 var3
       This is my prompt: Here are some words
    echo "  |$var1|$var2|$var3|"
       |Here|are|some words|
    ```
    ```

Example

- Let's put it all together
- We want an encrypt.sh script that accepts 4 arguments:
  - `./encrypt.sh -i inputfile -o outputfile`
- Our script should check to make sure the input is correct
- If everything checks out, we should encrypt the inputfile and write the output to outputfile
- Finally, it should return 0 if everything was successful

Example

- For starters, the `tr` command is used to translate characters from one to another
- We can use this command to employ the ROT13 cipher to "encrypt/decrypt" any input:

```
$ echo TEST | tr ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm
GRFG
$ echo GRFG | tr NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
TEST
```

Example

- It would be difficult to type out the tr command every time so let's construct a basic encrypt and decrypt Bash script:

```
[wbeldman@compute bin]$ pwd
/home/wbeldman/bin
[wbeldman@compute bin]$ ls -l
total 19
-rwxr-x--- 1 wbeldman wbeldman 121 Mar 19 15:51 decrypt.sh
-rwxr-x--- 1 wbeldman wbeldman 121 Mar 19 15:50 encrypt.sh
[wbeldman@compute bin]$ cat encrypt.sh
#!/bin/bash
tr ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm
[wbeldman@compute bin]$ cat decrypt.sh
#!/bin/bash
tr NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
[wbeldman@compute bin]$ echo TEST | ./encrypt.sh
GRFG
[wbeldman@compute bin]$ echo GRFG | ./decrypt.sh
TEST
[wbeldman@compute bin]$
```

Example

- Our script can accept an entire text file

```
[wbeldman@compute bin]$ ./encrypt.sh < ~/original-plaintext.txt > ~/original-ciphertext.txt
[wbeldman@compute bin]$ ./decrypt.sh < ~/original-ciphertext.txt > ~/decrypted-plaintext.txt
[wbeldman@compute bin]$ diff ~/original-plaintext.txt ~/decrypted-plaintext.txt
[wbeldman@compute bin]$ ls -l ~/original-plaintext.txt ~/decrypted-plaintext.txt
-rw------- 1 wbeldman wbeldman 3103 Mar 19 15:57 /home/wbeldman/decrypted-plaintext.txt
-rw------- 1 wbeldman wbeldman 3103 Mar 19 15:55 /home/wbeldman/original-plaintext.txt
[wbeldman@compute bin]$
```

Example

- Now we can expand our script
- Let's use if/else statements to accommodate arguments and react accordingly
- Let's prompt for confirmation
- Let's use a for loop to add a countdown timer
- (The decryption script would be the same as the encryption script, just with a different tr command and some wording changes)

Example

```
#!/bin/bash
echo "Welcome to $0!";
echo "You supplied $# arguments";
echo "The arguments are: $@";

if [[ $# != 4 ]]; then
    echo "Proper usage is $0 -i inputfile -o outputfile";
    exit 1;
fi;

input="";   #This will be our inputfile
output="";  #This will be our outputfile

# (Allow either order)
if [[ $1 == "-i" && $3 == "-o" ]]; then
    input=$2;
    output=$4;
elif [[ $1 == "-o" && $3 == "-i" ]]; then
    input=$4;
    output=$2;
else
    echo "Proper usage is $0 -i inputfile -o outputfile";
    exit 1;
fi;
```

Example (continued)

```
echo "Ready to encrypt ${input} to ${output}";
read -p "Press y to continue: " usercommand;
if [[ ${usercommand} != "y" && ${usercommand} != "Y" ]]; then
    echo "Cancelling because you said ${usercommand}";
    exit 1;
fi;

echo -n "Encrypting in "
for i in 3 2 1; do
    echo -n "$i ";
    sleep 1;    #Pause for 1 second
done;
echo

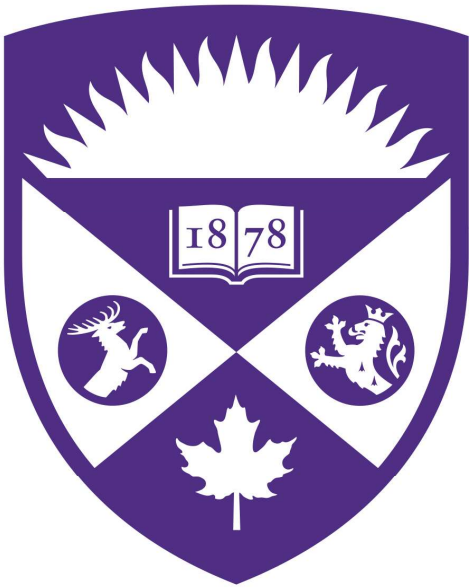
#Split this long command over multiple lines with a backslash
tr ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz \
   NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm \
   < $input \
   > $output

if [[ $? == 0 ]]; then
    echo "Encryption done!";
else
    echo "Encryption failed!";
    exit 1;
fi;

exit 0;
```

More information

- Shell scripting is a big topic
- Bash supports many advanced programming topics (e.g. arrays, functions/subroutines)
- Consult the man page for bash or other resources online for more complicated options
- https://devhints.io/bash



Western

UNIVERSITY · CANADA