P2T: C Programming under Linux

Bash Scripting Guide

1. Simple scripts

```
#! /bin/bash
# Example of an extremely simple script

echo "Welcome to Bash!"

exit 0   # Success
```

The line **#! /bin/bash** must appear at the start of all Bash scripts. Comments are preceded with the hash (**#**) character, which can occur either at the start of a line, or in the middle.

You can include any commands that you would type into the terminal in a script. For instance, you can display information by using the **echo** command.

When a Bash script exits, a numerical code is returned which can be used to check what happened. An exit code of **0** means everything worked correctly, while a non-zero code means something went wrong. Some values have special meanings (e.g. **130** means that **CTRL + C** was used to end the script) but in general you can use non-zero values as you wish to identify different problems[1]. A value of **0** is returned by default, but it is still a good idea to explicitly include **exit 0** at the end of your scripts.

2. Layout and comments

As with the C exercises, you should try to lay your code out neatly, as this makes it much easier to read and understand what is going on, particularly when your scripts get longer. You should also include comments where necessary to explain what your code does; at the very least, a short comment immediately after the interpreter directive saying what the script does is a good idea. Comments in Bash begin with **#**.

Bash is mostly fairly happy for you to use white space as you wish, and we encourage you to use blank lines and indentation to lay your scripts out neatly. There are a few situations where Bash is fussy about white space, including before and after the square brackets in conditional tests.

In later exercises and the practical exam, a number of the available marks are awarded specifically for layout and comments.

[1] See https://tldp.org/LDP/abs/html/exitcodes.html for a list of standard exit codes.

3. Branching: conditional statements

```
#! /bin/bash
# Example of a conditional statement

if [[ -f "input.txt" ]]; then
    echo "Input file located"
elif [[ -f "alternative_input.txt" ]]; then
    echo "Alternative input file located"
else
    echo "ERROR: No input file located"
    exit 1  # This is an error
fi

exit 0  # Success
```

Bash has an **`if … elif … else … fi`** structure which can be used to branch within your scripts. Both the **`elif`** and **`else`** sections are optional. The conditional test is included within doubled square brackets: **`[[ ]]`**. Some of the most common tests are listed below.

4. Tests

String tests	**Result**
`[[ STRING1 = STRING2 ]]`	True if the strings are equal
`[[ STRING1 != STRING2 ]]`	True if the strings are not equal
`[[ -n STRING ]]`	True if the length of the string is not zero
`[[ -z STRING ]]`	True if the length of the string is zero
Arithmetic tests	**Result**
`[[ A -eq B ]]`	True if the expressions are equal
`[[ A -ne B ]]`	True if the expressions are not equal
`-gt`, **`-ge`**, **`-lt`**, **`-le`**	Used as above: greater than (**`-gt`**), greater than or equal to (**`-ge`**), less than (**`-lt`**), and less than or equal to (**`-le`**)
`[[ !A ]]`	Inverts the expression: true if the expression is false, or false if the expression is true
File tests	**Result**
`[[ -d FILE ]]`	True if the file is a directory
`[[ -e FILE ]]`	True if the file exists

[[-f FILE]]	True if the file is a regular file
[[-r FILE]]	True if the file is readable
[[-s FILE]]	True if the file has a non-zero size (i.e. it is not empty)
[[-w FILE]]	True if the file is writable
[[-x FILE]]	True if the file is executable

You can combine tests using **&&** for "and" and **||** for "or".

5. **for** loops

Use a **for** loop when you want to repeat an action a certain number of times that you know in advance, or when you want to loop over a particular set of files. You can loop over text values like this:

```
for animal in "Rabbit" "Squirrel" "Otter"
do
    echo $animal
done
```

You can also loop over numerical ranges. You can save a little space by combining the loop declaration and keyword **do** on the same line, separated by a semicolon (**;**):

```
for number in {1..10}; do
    echo $number
done
```

If you want to use an increment other than 1, you can use a range like **{2..20..2}** (which will loop over the values **2**, **4**, **6**… **20**).

Finally, you can loop over files by pattern matching on their filenames:

```
# Loop over all files
for file in *; do
    echo $file
done

# Loop over text files with names beginning with the letter A
for textfile in A*.txt; do
    echo $textfile
done
```

6. **while** loops

Use a **while** loop when you want to repeat an action until something particular happens or until a condition is met. You will likely not know in advance how many times the loop is going to repeat. For example, you could pause a script until a particular file is created:

```
# While the results file does not exist...
while [[ ! -e results ]]; do
    # ...wait for ten seconds
    sleep 10
done
```

You can also use a **while** loop to loop over the contents of a file, as you will see in Section 10.

7. Variables

You can set and use variables in a script in the same you would from the terminal:

```
value=123.45
echo $value

# You can also access environment variables within your script
echo "You are $USER working on $HOSTNAME"
```

Special variables exist to let you access useful properties of scripts, including their command-line arguments:

Variable name	Value
$0	The name of the script
$#	The number of parameters or arguments passed to the script
$1	The first parameter passed to the script (and **$2** is the second, etc.)
$@	All of the parameters passed to the script, which is useful for looping over
$$	The process ID of the script
$?	The exit code of the last command

Unlike their equivalents in C, **$#** and **$@** do not count the name of the script as the first argument. You can use the above variables to check that the correct arguments have been provided or to see whether the commands you are running are working:

```
#! /bin/bash

# Check that the correct number of arguments has been provided
if [[ $# -ne 1 ]]; then
    echo "You must provide the name of a file"
    exit 1
fi

# Run a simple command with the argument, and check its exit code
ls -l $1
if [[ $? -ne 0 ]]; then
    echo "Something went wrong with the ls command"
    exit 2
fi

exit 0
```

8. Command substitution

Command substitution lets you take the output of a command and store it in a variable. The recommended syntax uses parentheses:

```
result=$(ls -l . | wc -l)
```

An older syntax using backticks (`) also exists, and you may come across examples of this style online:

```
result=`ls -l . | wc -l`
```

9. Interactive input

You can get input from the user and store it in a variable using the **read** command:

```
echo "Enter a value: "
read value
echo "You entered ${value}"
```

You can also combine the prompt with the **read** command like this:

```
read -p "Enter another value: " value
echo "You entered ${value}"
```

10. Reading files

The easiest way to read the contents of a file one line at a time is to redirect standard input to the **read** command as part of a **while** loop:

```
# Read file called FILENAME and store each line in variable $line
while read line; do
    echo $line
done < FILENAME
```

11. Functions

A function definition comprises a name followed by parentheses (**()**), and then the body of the function contained within braces (**{}**):

```
print_username() {
    echo "You are user $USER"
}
```

You can call a function using just its name; you do not need to include the parentheses:

```
# Call the print_username function
print_username
```

You can pass parameters to a function, and refer to these within the function using the variables **$1**, **$2**, etc.:

```
print_greeting() {
    echo "Hello ${1}!"
}

# Call the print_greeting function, passing it a single parameter
print_greeting Gordon
```

12. Next steps

The Linux Documentation Project provides both introductory and advanced guides to Bash scripting:

https://tldp.org/LDP/Bash-Beginners-Guide/html/Bash-Beginners-Guide.html

https://tldp.org/LDP/abs/html/index.html

If you are looking for more information on Bash, these comprehensive guides are well worth consulting.