The Bash Shell and Basic Scripting

A shell is simply a command line interpreter which provides the user interface for terminal windows. A command shell can also be used to run scripts, even in non-interactive sessions without a terminal window, as if the commands were being directly typed in.

`sh` was written by Steve Bourne at AT&T in 1977, and is often known as the Bourne Shell. `bash` is a product of the GNU project and was created in 1987. It was designed as a major upgrade of `sh`; the name stands for Bourne Again Shell. On all Linux systems `sh` is just a link to `bash`, but scripts which are invoked as `sh` will only work without the `bash` extensions.

Linux provides a wide choice of shells; exactly what is available on the system is listed in the `/etc/shells` file.

A Simple `bash` Script

The first line of the script, `#!/usr/bin/env bash`, gets the path of the command interpreter from the system environment. The special two-character sequence, `#!`, is often called a shebang.

Save it as `hello.sh` and enter `chmod +x hello.sh` to make the file executable by all users:

```
#!/usr/bin/env bash
echo "Hello Linux Foundation Student"
```

You can then run the script by typing `./hello.sh` or by doing:

```
$ bash hello.sh
```

Interactive Example Using bash Scripts

Create a file named `getname.sh` in your favorite editor with the following content:

```
#!/usr/bin/env bash
echo "ENTER YOUR NAME"
read name

# NOT RECOMMENDED:
# echo The name given was: $name
echo "The name given was: ${name}"
```

The user will be prompted to enter a value, which is then displayed on the screen. We can reference the value of a shell variable by using a `$` in front of the variable name, such as `$name`. However, the preferred approach is `"${name}"` because it is more robust in scenarios like spaces in the value.

Return Values

All shell scripts generate a return value upon finishing execution. By convention, success is returned as zero, and failure is returned as any non-zero value. The return value is stored in the environment variable represented by `$?`:

```
$ ls /etc/logrotate.conf
/etc/logrotate.conf

$ echo $?
0
```

In this example, the system is able to locate the file `/etc/logrotate.conf` and `ls` returns a value of 0 to indicate success. When run on a non-existing file, it returns 2.

Splitting Long Commands Over Multiple Lines

The concatenation operator (`\`), the backslash character, is used to continue long commands over several lines. It causes the shell to combine (concatenate) multiple lines and execute them as one single command. Here is an example of a command installing a long list of packages on a system using Debian package management:

```
$ sudo apt install autoconf automake bison build-essential \
  chrpath curl diffstat emacs flex gcc-multilib g++-multilib \
  libsdl1.2-dev libtool lzop make mc patch \
  screen socat sudo tar texinfo tofrodos u-boot-tools unzip \
  vim wget xterm zip
```

Note that the backslash character has another meaning of interpreting next character literally:

```
#!/usr/bin/env bash
echo \$HOME # $HOME
```

Putting Multiple Commands on a Single Line

Users sometimes need to combine several commands and statements and even conditionally execute them based on the behavior of operators used in between them. This method is called chaining of commands.

The `;` (semicolon) character is used to separate the commands and execute them sequentially, as if they had been typed on separate lines. Each ensuing command is executed whether or not the preceding one succeeded. Thus, the three commands in the following example will all execute, even if the ones preceding them fail: `make ; make install ; make clean`.

However, you may want to abort subsequent commands when an earlier one fails. You can do this using the `&&` (and) operator as in: `make && make install && make clean`.

Likewise, the `||` (or) operator runs through multiple commands until something succeeds: `cat file1 || cat file2 || cat file3`.

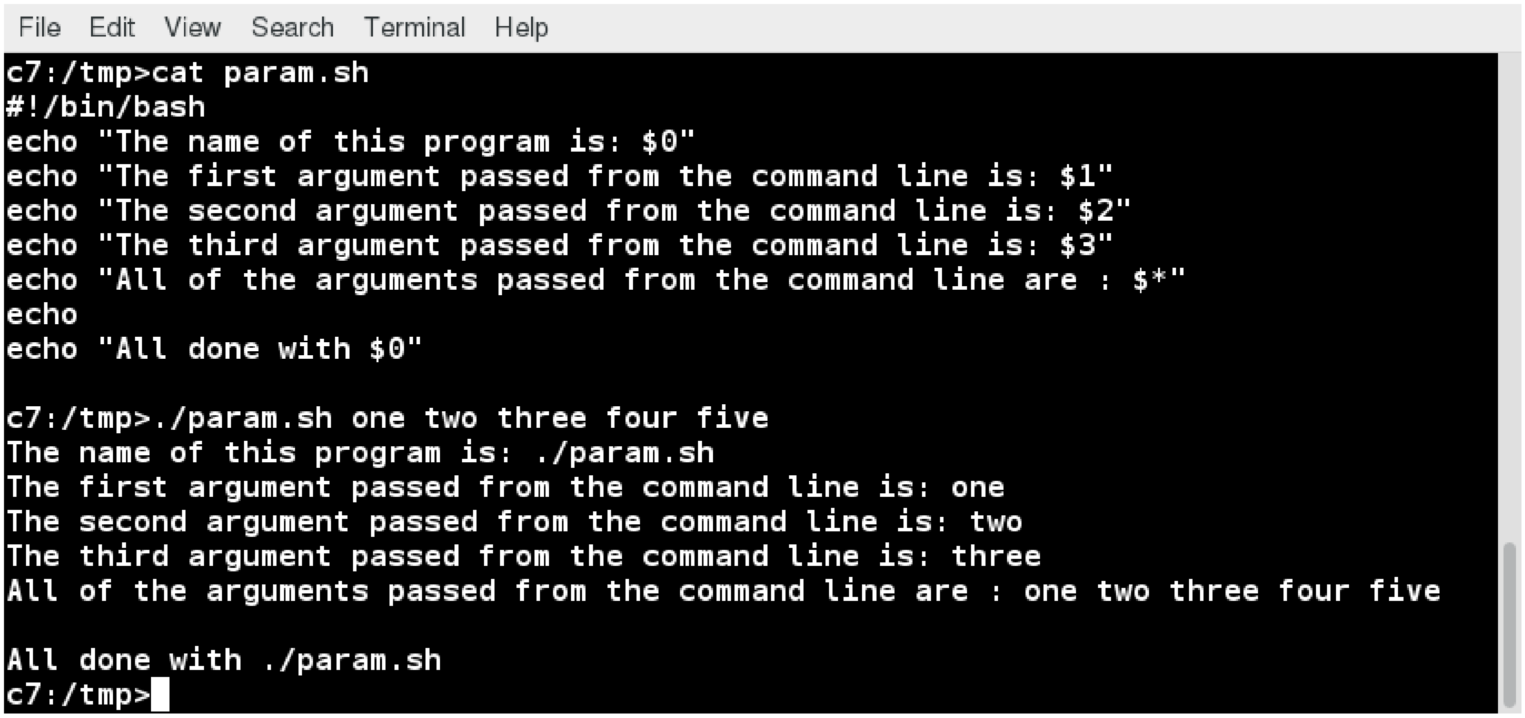
Script Parameters

Users often need to pass parameter values to a script:

```
$ ./script.sh /tmp
$ ./script.sh 100 200
```

Within a script, the parameter or an argument is represented with a `$` and a number or special character:

Parameter	Meaning
`$0`	Script name
`$1`	First parameter
`$2`, `$3`, etc.	Second, third parameter, etc.
`$*`	All parameters
`$#`	Number of arguments



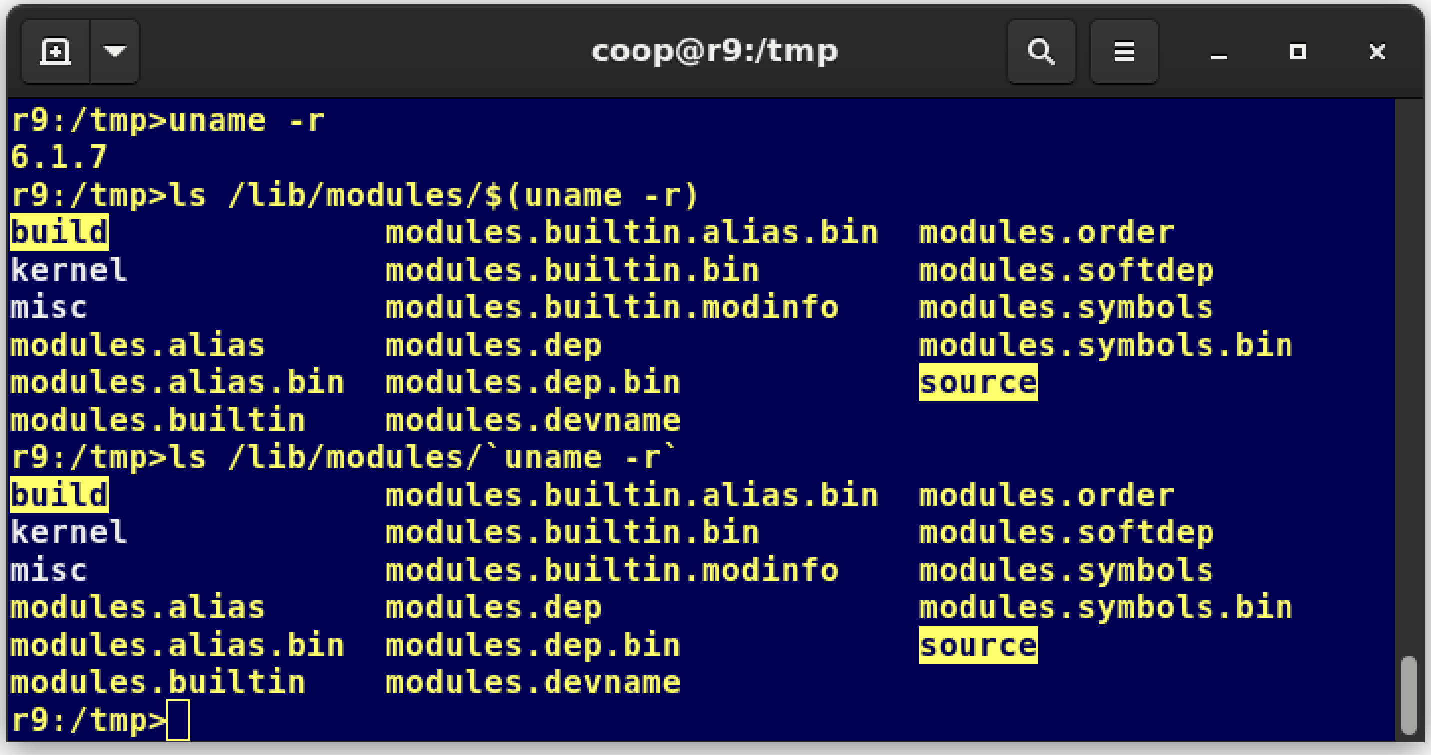
```
c7:/tmp>cat param.sh
#!/bin/bash
echo "The name of this program is: $0"
echo "The first argument passed from the command line is: $1"
echo "The second argument passed from the command line is: $2"
echo "The third argument passed from the command line is: $3"
echo "All of the arguments passed from the command line are : $*"
echo
echo "All done with $0"

c7:/tmp>./param.sh one two three four five
The name of this program is: ./param.sh
The first argument passed from the command line is: one
The second argument passed from the command line is: two
The third argument passed from the command line is: three
All of the arguments passed from the command line are : one two three four five

All done with ./param.sh
c7:/tmp>
```

Command Substitution

At times, you may need to substitute the result of a command as a portion of another command. It can be done in two ways:

- By enclosing the inner command in `$()`
- By enclosing the inner command with backticks (`` ` ``)

The second form using backticks is deprecated, and its use should be avoided in new scripts and commands. No matter which method is used, the specified command will be executed in a newly launched shell environment, and the standard output of the shell will be inserted where the command substitution is done.



```
r9:/tmp>uname -r
6.1.7
r9:/tmp>ls /lib/modules/$(uname -r)
build              modules.builtin.alias.bin  modules.order
kernel             modules.builtin.bin        modules.softdep
misc               modules.builtin.modinfo    modules.symbols
modules.alias      modules.dep                modules.symbols.bin
modules.alias.bin  modules.dep.bin            source
modules.builtin    modules.devname
r9:/tmp>ls /lib/modules/`uname -r`
build              modules.builtin.alias.bin  modules.order
kernel             modules.builtin.bin        modules.softdep
misc               modules.builtin.modinfo    modules.symbols
modules.alias      modules.dep                modules.symbols.bin
modules.alias.bin  modules.dep.bin            source
modules.builtin    modules.devname
r9:/tmp>
```

Environment Variables

When referenced, environment variables must be prefixed with the `$` symbol, as in `$HOME`. You can view and set the value of environment variables. However, no prefix is required when setting or modifying the variable value: `MYCOLOR=blue`.

By default, the variables created within a script are available only to the subsequent steps of that script. Any child processes (sub-shells) do not have automatic access to the values of these variables. To make them available to child processes, they must be promoted to environment variables using the export statement: `export VAR=value`.

While child processes are allowed to modify the value of exported variables, the parent will not see any changes; exported variables are not shared, they are only copied and inherited.

Functions

A function is a code block that implements a set of operations. They are also often called subroutines.

```
showmess() {
    echo "My favorite Linux distribution is: $1"
}

# My favorite Linux distribution is: Ubuntu
# My favorite Linux distribution is: Fedora
# My favorite Linux distribution is: Debian
# My favorite Linux distribution is: openSUSE
showmess Ubuntu
showmess Fedora
showmess Debian
showmess openSUSE
```

Boolean Expressions

Operator	Operation
&&	AND
||	OR
!	NOT

The if Statement

When an if statement is used, the ensuing actions depend on the evaluation of specified conditions.

In the following example, an if statement checks to see if a certain file exists, and if the file is found, it displays a message indicating success or failure:

```
if [ -f "$1" ]
then
    echo file "$1" exists
else
    echo file "$1" does not exist
fi
```

Notice the use of the square brackets ([]) to delineate the test condition.

In modern scripts, you may see doubled brackets as in `[[ -f /etc/passwd ]]`. This is not an error. It is never wrong to do so and it avoids some subtle problems, such as referring to an environment variable without surrounding it in double quotes.

The elif Statement

You can use the elif statement to perform more complicated tests, and take action appropriate actions.

```
give_your_name() {
    echo "Give your name"
    read name

    if [[ "$name" == "John" ]]; then
        echo "Hello John"
    elif [[ "$name" == "George" ]] || [[ "$name" == "Ringo" ]] || [[ "$name" == "Paul" ]];
then
        echo "Hello $name"
    else
        echo "Forget it $name, you are not a Beatle"
    fi
}

# Give your name
# Hello John
```

```
echo "John" | give_your_name

# Give your name
# Hello George
echo "George" | give_your_name

# Give your name
# Hello Jack
echo "Jack" | give_your_name
```

Testing for Files

bash provides a set of file conditionals that can be used with the if statement. In the following example,

```
# Note the very common practice of putting then on the same
# line as the if statement.
if [ -x /etc/passwd ]; then
    echo "Executable"
fi
```

the if statement checks if the file `/etc/passwd` is executable, which it is not.

Condition	Meaning
`-e file`	Checks if the file exists
`-d file`	Checks if the file is a directory
`-f file`	Checks if the file is a regular file (i.e., not a symbolic link, device node, directory, etc.)
`-s file`	Checks if the file is of non-zero size
`-r file`	Checks if the file is readable
`-w file`	Checks if the file is writable
`-x file`	Checks if the file is executable

Testing for Strings

You can use the if statement to compare strings using the operator `==` (two equal signs):

```
test_string() {
    echo "Please specify window, middle, or aisle for your seat."
    read choice

    if [[ "$choice" == "window" ]]; then
        echo "Window #29A"
    elif [[ "$choice" == "middle" ]]; then
        echo "Middle #29B"
    elif [[ "$choice" == "aisle" ]]; then
        echo "Aisle #29C"
    else
        echo "\"$choice\" is not valid. Please try again."
    fi
}

# Please specify window, middle, or aisle for your seat.
# Middle #29B
echo "middle" | test_string

# Please specify window, middle, or aisle for your seat.
# Window #29A
echo "window" | test_string

# Please specify window, middle, or aisle for your seat.
# "wing" is not valid. Please try again.
echo "wing" | test_string
```

Note that using one `=` sign will also work, but some consider it deprecated usage.

Testing for Numbers

Operator	Meaning
`-eq`	Equal to
`-ne`	Not equal to
`-gt`	Greater than
`-lt`	Less than
`-ge`	Greater than or equal to
`-le`	Less than or equal to

```
check_age() {
    AGE=$1
    if [[ $AGE -ge 20 ]] && [[ $AGE -lt 30 ]]; then
        echo "You are in your 20s"
    elif [[ $AGE -ge 30 ]] && [[ $AGE -lt 40 ]]; then
        echo "You are in your 30s"
    elif [[ $AGE -ge 40 ]] && [[ $AGE -lt 50 ]]; then
        echo "You are in your 40s"
    else
        echo "You are not in the proper range of 21-50"
    fi
}

check_age 33    # You are in your 30s
check_age 21    # You are in your 20s
check_age 18    # You are not in the proper range of 21-50
```

Arithmetic Expressions

Arithmetic expressions can be evaluated in the following three ways (spaces are important!):

- `expr` is a standard but somewhat deprecated program

  ```
  echo $(expr 8 + 8)
  ```

- Using the `$((...))` syntax, the built-in shell format
  - In modern shell scripts, the use of `expr` is better replaced with `x=$((...))`

  ```
  echo $((x+1))
  ```

- Using the built-in shell command `let`

  ```
  let x=( 1 + 2 ); echo $x
  ```