Software I: Utilities and Internals

Lecture 8 – Shell Programming Using **`bash`**

Steps in Writing a Shell Script

- The first line identifies the file as a **`bash`** script.

  ```
  #!/bin/bash
  ```

- This is called the magic number; it identifies theprogam that should interpret the script.
- Comments begin with a **`#`** and end at the end of the line.
- Use chmod to give the user (and others, if (s)he wishes) permission to execute it.

Running **myscript**

```
SIEGFRIE@panther:~$ chmod +x bin/myscript
SIEGFRIE@panther:~$ myscript
Hello SIEGFRIE, it's nice talking to you.
Your present working directory is /home/siegfried.
You are working on a machine called
panther.adelphi.edu.
Here is a list of your files.
160L2Handout.pdf    data          j        save
160l4notes.pdf      data.dat      java     script.sed
… …          … …                … …
CS271               HW2.doc       recipes  CSC 390.7z
index.html          sample.c
Bye for now SIEGFRIE. the time is 18:51:45!
```

myscript

```
#!/bin/bash
# This is the first Bash shell program of the day.
# Scriptname: greetings
# Brian Bashful
echo "Hello $LOGNAME, it's nice talking to you."
echo "Your present working directory is `pwd`."
echo "You are working on a machine called `uname -n`."
echo "Here is a list of your files."
ls      # List file in the present working directory
echo "Bye for now $LOGNAME. the time is `date +%T`!"
```

The **`read`** Command

- The read command is a built-in command used to read input from the terminal or from a file.
- The read command takes a line of input until a newline (which is translated as a NULL byte (**`'\0'`**) .
- It will save the input in a shell variable **`REPLY`** unless you specify where to save it.

The **`read`** Command (continued)

Format	Meaning
`read answer`	Reads a line from **`stdin`** into the variable **`answer`**
`read first last`	Reads a line from **`stdin`** up to the whitespace, putting the first word in **`first`** and the rest of the of line into **`last`**
`read`	Reads a line from **`stdin`** and assigns it to **`REPLY`**
`read -a arrayname`	Reads a list of word into an array called **`arrayname`**
`read -p prompt`	Prints a prompt, waits for input and stores input in **`REPLY`**
`read -r line`	Allows the input to contain a backslash.

bin/nosy

```
#!/bin/bash
#Scriptname: nosy

echo -e "Are you happy?\c"
read answer
echo "$answer is the right response."
echo -e "What is your full name? \c"
read first middle last
echo "Hello, $first"
echo -n "Where do you work? "
read
echo I guess $REPLY keeps you busy!
```

```
read -p "Enter your job title: "
echo "I though you might be an $REPLY."
echo -n "Who are your best friends? "
read -a friends
echo "Say hi to ${friends[2]}."
```

Running **nosy**

```
SIEGFRIE@panther:~$ nosy
Are you happy?Yes
Yes is the right response.
What is your full name? Robert Michael Siegfried
Hello, Robert
Where do you work? Adelphi University
I guess Adelphi University keeps you busy!
Enter your job title: Professor
I though you might be an Professor.
Who are your best friends? David Elliot Marvin Kathy
Say hi to Marvin.
SIEGFRIE@panther:~$
```

The **declare** Command

- The **declare** command (with the **-i** option) allows the user to declare a shell variable as an integer.
- Assigning a **float** value to such a variable is a syntax error.
- This allows shell variables to be used for arithmetic.

declare Statement – An Example

```
SIEGFRIE@panther:~$ declare -i num
SIEGFRIE@panther:~$ num=hello
SIEGFRIE@panther:~$ echo $num
0
SIEGFRIE@panther:~$ num=5 + 5
+: command not found
SIEGFRIE@panther:~$ num=5+5
SIEGFRIE@panther:~$ echo $num
10
SIEGFRIE@panther:~$ num=6*4
SIEGFRIE@panther:~$ echo $num
24
```

```
SIEGFRIE@panther:~$ num=6.5
-bash: 6.5: syntax error: invalid arithmetic
operator (error token is ".5")
SIEGFRIE@panther:~$
```

Listing Integer Variables

```
SIEGFRIE@panther:~$ declare -i
declare -ir BASHPID
declare -ir EUID="16131"      # Effective user id
declare -i HISTCMD            # History list index
declare -i LINENO
declare -i MAILCHECK="60"
declare -i OPTIND="1"   #Holds the index to the next
                        # argument to be processed
declare -ir PPID="27414"      # Parent Process id
declare -i RANDOM       # Produce random integer
declare -ir UID="16131"       # user ID
declare -i num="24"
SIEGFRIE@panther:~$
```

Representing Integers in Different Bases

- Numbers can be represented in decimal, octal, hexadecimal and and binary.

Integers in Other Bases - Examples

```
SIEGFRIE@panther:~$ declare -i n
SIEGFRIE@panther:~$ n=2#101
SIEGFRIE@panther:~$ echo $n
5
SIEGFRIE@panther:~$ declare -i x=017
SIEGFRIE@panther:~$ echo $x
15
SIEGFRIE@panther:~$ x=2#101
SIEGFRIE@panther:~$ echo $x
5
SIEGFRIE@panther:~$ x=8#17
SIEGFRIE@panther:~$ echo $x
15
```

```
SIEGFRIE@panther:~$ x=16#b
SIEGFRIE@panther:~$ echo $x
11
SIEGFRIE@panther:~$ x=0x17
SIEGFRIE@panther:~$ echo $x
23
SIEGFRIE@panther:~$
```

Floating Point Arithmetic

- **Bash** does not support floating point arithmetic but **bc**, **awk** and **nawk** utilities all do.

```
SIEGFRIE@panther:~$ n=`echo "scale=3; 13 / 2" | bc`
SIEGFRIE@panther:~$ echo $n
6.500
SIEGFRIE@panther:~$ product=`nawk -v x=2.45 -v
y=3.123 'BEGIN{printf "%.2f\n", x*y}'`
SIEGFRIE@panther:~$ echo $product
7.65
SIEGFRIE@panther:~$
```

Positional Parameters

Positional Parameter	What It References
`$0`	References the name of the script
`$#`	Holds the value of the number of positional parameters
`$*`	Lists all of the positional parameters
`$@`	Means the same as $@, except when enclosed in double quotes
`"$*"`	Expands to a single argument (e.g., "`$1 $2 $3`")
`"$@"`	Expands to separate arguments (e.g., "`$1`" "`$2`" "`$3`")
`$1 .. ${10}`	References individual positional parameters

Positional Parameters – An Example

```
SIEGFRIE@panther:~$ cat bin/greetings2
#!/bin/bash
#Scriptname: greetings2
echo "This script is called $0."
echo "$0  $1  and $2"
echo "The number of positional parameters is $#"
SIEGFRIE@panther:~$ greetings2
This script is called
/home/siegfried/bin/greetings2.
/home/siegfried/bin/greetings2    and
The number of positional parameters is 0

SIEGFRIE@panther:~$
```

Positional Parameters – An Example

```
SIEGFRIE@panther:~$ greetings2 Tommy
This script is called
/home/siegfried/bin/greetings2.
/home/siegfried/bin/greetings2  Tommy  and
The number of positional parameters is 1
SIEGFRIE@panther:~$ greetings2 Tommy Kimberly
This script is called
/home/siegfried/bin/greetings2.
/home/siegfried/bin/greetings2  Tommy  and Kimberly
The number of positional parameters is 2
SIEGFRIE@panther:~$
```

The **`set`** Command

- The set command with arguments resets the positional parameters.
- Once reset, the old parameter list is lost.
- To unset all of the positional parameters, use
  `set <values1> <value2>`
- **`$0`** is always the name of the script.

`set` – An Example

```
#!/bin/bash
# Scriptname: args
# Script to test command-line parameters
echo The name of this script is $0.
echo The arguments are $*.
echo The first argument is $1.
echo The second argument is $2.
echo The number of arguments is $#.
oldargs=$*
set Jake Nicky Scott# Reset the positional parameters
echo All the positional parameters are $*.
echo "Goodbye for now, $1."
set $(date)    #Reset the positional parameters
echo The date is $2 $3, $6.
echo "The value of \$oldargs is $oldargs."
set $(date)
echo $1 $2 $3
```

```
SIEGFRIE@panther:~$ args a b c d
The name of this script is /home/siegfried/bin/args.
The arguments are a b c d.
The first argument is a.
The second argument is b.
The number of arguments is 4.
All the positional parameters are Jake Nicky Scott.
Goodbye for now, Jake.
The date is Oct 22, 2013.
The value of $oldargs is a b c d.
Tue Oct 22
SIEGFRIE@panther:~$
```

set – Another Example

```
SIEGFRIE@panther:~$ cat bin/checker
#!/bin/bash
# Scriptname: checker
# Script to demonstrate the use of special variable
modifiers and argumetns
name=${1:?"requires an argument" }
echo Hello $name
SIEGFRIE@panther:~$ checker
/home/siegfried/bin/checker: line 4: 1: requires an
argument
SIEGFRIE@panther:~$ checker sue
Hello sue
SIEGFRIE@panther:~$
```

$* vs $@

```
SIEGFRIE@panther:~$ set 'apple pie' pears peaches
SIEGFRIE@panther:~$ for i in $*
> do
> echo $i
> done
apple
pie
pears
peaches
SIEGFRIE@panther:~$ set 'apple pie' pears peaches
SIEGFRIE@panther:~$ for i in "$*"
> do
> echo $i
> done
apple pie pears peaches
SIEGFRIE@panther:~$
```

```
SIEGFRIE@panther:~$ set 'apple pie' pears peaches
SIEGFRIE@panther:~$ for i in $@
> do
> echo $i
> done
apple
pie
pears
peaches
SIEGFRIE@panther:~$ set 'apple pie' pears peaches
SIEGFRIE@panther:~$ for i in "$@"
> do echo $i
> done
apple pie
pears
peaches
SIEGFRIE@panther:~$
```

Exit Status

- Every process running in Linux has an exit status code, where **0** indicates successful conclusion of the process and nonzero values indicates failure to terminate normally.
- Linux and UNIX provide ways of determining an exit status and to use it in shell programming.
- The **?** in **bash** is a shell variable that contains a numeric value representing the exit status.

Exit Status – An Example

```
SIEGFRIE@panther:~$ name=whoopsie
SIEGFRIE@panther:~$ grep  $name /etc/passwd
whoopsie:x:104:107::/nonexistent:/bin/false
SIEGFRIE@panther:~$ echo $?
0  ←———— Success
SIEGFRIE@panther:~$ name="Tom"
SIEGFRIE@panther:~$ grep  $name /etc/passwd
SIEGFRIE@panther:~$ echo $?
1  ←———— Failure
SIEGFRIE@panther:~$
```

`test` Command

- The **`test`** command is used to evaluate an expression, usually to determine a boolean condition.
- The **`test`** command or an expression in brackets is used.
- Shell metacharacters are not expanded with **`test`** or brackets.
- Strings that contain whitespace must be in quotes.

`test` Command Operators – String Test

Test Operator	Tests True if
`[ string1 = string2 ]`	String1 is equal to String2 (space surrounding = is necessary
`[ string1 != string2 ]`	String1 is not equal to String2 (space surrounding != is not necessary
`[ string ]`	String is not null.
`[ -z string ]`	Length of string is zero.
`[ -n string ]`	Length of string is nonzero.
`[ -l string ]`	Length of string (number of character)

`test` Command Operators – Logical Tests

Test Operator	Test True If
`[ string1 -a string2 ]`	Both string1 and string 2 are true.
`[ string1 -o string2 ]`	Both string1 or string 2 are true.
`[ ! string ]`	Not a string1 match

Test operator	Tests True if
`[[ pattern1 && Pattern2 ]]`	Both pattern1 and pattern2 are true
`[[ pattern1 || Pattern2 ]]`	Either pattern1 or pattern2 is true
`[[ !pattern ]]`	Not a pattern match

pattern1 and *pattern2* can contain metacharacters.

`test` Command Operators – Integer Tests

Test operator	Tests True if
`[ int1 -eq int2 ]`	int1 = int2
`[ int1 -ne int2 ]`	int1 ≠ int2
`[ int1 -gt int2 ]`	int1 > int2
`[ int1 -ge int2 ]`	int1 ≥ int2
`[ int1 -lt int2 ]`	int1 < int2
`[ int1 -le int2 ]`	int1 ≤ int2

`test` Command Operators – File Tests

Test Operator	Test True If
`[ file1 -nt file2 ]`	True if file1 is newer than file2*
`[ file1 -ot file2 ]`	True if file1 is older than file2*
`[ file1 -ef file2 ]`	True if file1 and file2 have the same device and inode numbers.

* according to modfication date and time

`test` – Example

```
SIEGFRIE@panther:~$ test $name != whoopsie
SIEGFRIE@panther:~$ echo $?
1
SIEGFRIE@panther:~$ [ $name = Tom ]
SIEGFRIE@panther:~$ echo $?
1
SIEGFRIE@panther:~$ [ $name = whoopsie ]
SIEGFRIE@panther:~$ echo $?
0
SIEGFRIE@panther:~$ [ $name = [Ww]??????? ]
SIEGFRIE@panther:~$ echo $?
1
SIEGFRIE@panther:~$
```

```
SIEGFRIE@panther:~$ x=5
SIEGFRIE@panther:~$ y=20
SIEGFRIE@panther:~$ [ $x -gt $y ]
SIEGFRIE@panther:~$ echo $?
1
SIEGFRIE@panther:~$ [ $x -lt $y ]
SIEGFRIE@panther:~$ echo $?
0
SIEGFRIE@panther:~$
```

```
SIEGFRIE@panther:~$ name=Tom; friend=Joseph
SIEGFRIE@panther:~$ [[ $name == [Tt]om ]]
SIEGFRIE@panther:~$ echo $?
0
SIEGFRIE@panther:~$ [[ $name == [Tt]om && $friend ==
"Jose" ]]
SIEGFRIE@panther:~$ echo $?
1
SIEGFRIE@panther:~$ shopt -s extglob   # Turns on
extended pattern matching
SIEGFRIE@panther:~$ name=Tommy
SIEGFRIE@panther:~$ [[ $name == [Tt]o+(m)y ]]
SIEGFRIE@panther:~$ echo $?
0
SIEGFRIE@panther:~$
```

`let` and arithmetic with `(( ))`

- The **`let`** command allows the user to perform arithmetic with a large set of operators.
- Testing an expression with the **`let`** command, **`test`** command, or double parentheses produce **`0`** or false and nonzero for true.

`(( ))` – An Example

```
SIEGFRIE@panther:~$ x=2
SIEGFRIE@panther:~$ y=3
SIEGFRIE@panther:~$ (( x > 2 ))
SIEGFRIE@panther:~$ echo $?
1
SIEGFRIE@panther:~$ (( x < 2 ))
SIEGFRIE@panther:~$ echo $?
1
SIEGFRIE@panther:~$ ((x ==2 && y == 3 ))
SIEGFRIE@panther:~$ echo $?
0
SIEGFRIE@panther:~$ (( x > 2 || y < 3 ))
SIEGFRIE@panther:~$ echo $?
1
SIEGFRIE@panther:~$
```

The `if` Command

- **`if`** is the simplest conditional statement in **`bash`**.
- **`if`** uses the exit status returned by a program; if the exit status is `0`, it executes the then clause.
- It is important to know what the exit status is so that it can be used correctly.

Format of the `if` command

```
if command
then
     command
     command
fi
```

Using **`test`** For Numbers And Strings – Old Format

```
if test expression
then
     command
fi
```

or

```
if [ string/numeric expression] then
     command
fi
```

Using **`test`** For Strings – New Format

```
if [[ string expression ]]  then
     command
fi
```

or

```
if (( numeric expression ))
```

if – An Example

```
SIEGFRIE@panther:~$ if grep "$name" /etc/passwd >
/dev/null 2>&1
> then
>    echo Found $name
> fi
Found
SIEGFRIE@panther:~$
```

- **grep** searches for name in the password file. **stdout** and **stderr** are redirected to **/dev/null**, the Linux bit bucket.

if – Another Example

```
SIEGFRIE@panther:~$ cat bin/ok2
echo "Are you o. k. (y/n) ?"
read answer
if [ $answer = Y –o "$answer" = y ]
then
        echo "Glad to hear it."
fi

SIEGFRIE@panther:~$ ok2
Are you o. k. (y/n) ?
Y
Glad to hear it.
SIEGFRIE@panther:~$
```

```
SIEGFRIE@panther:~$ ok2
Are you o. k. (y/n) ?
y
Glad to hear it.
SIEGFRIE@panther:~$ ok2
Are you o. k. (y/n) ?
Yes,you betcha
/home/siegfried/bin/ok2: line 3: [: too many
arguments
SIEGFRIE@panther:~$
```

if – Another Example

```
SIEGFRIE@panther:~$ cat bin/ok3
echo "Are you o. k. ?"
read answer
if [[ $answer == [Yy]* || $answer == Maybe ]]
then
        echo "Glad to hear it."
fi
SIEGFRIE@panther:~$ ok3
Are you o. k. ?
y
Glad to hear it.
SIEGFRIE@panther:~$
```

```
SIEGFRIE@panther:~$ ok3
Are you o. k. ?
Y
Glad to hear it.
SIEGFRIE@panther:~$ ok3
Are you o. k. ?
Yes, you betcha
Glad to hear it.
SIEGFRIE@panther:~$
```

`exit` Command and the `?` Variable

- **`exit`** is used to terminate the script; it is mainly to used to exit the script if some condition is true.
- **`exit`** has one parameter – a number ranging from **`0`** to **`255`**, indicating if is ended successfully (**`0`**) or unsuccessfully (nonzero).
- The argument given to the script is stored in the variable **`?`**

bin/bigfiles

```
SIEGFRIE@panther:~$ cat bin/bigfiles
#!/bin/bash
# Name: bigfiles
# Purpose: Use the find command to find any files in
# the root partition that have not been modified
# within the past n (any number within 30 days) days
# and are larger than 20 blocks(512-byte blocks)

if (( $# != 2 ))  # [ $# -ne 2 ]
then
        echo "Usage: $0 mdays size " 1>&2
        exit 1
fi
```

```
if (( $1 < 0 || $1 > 30 ))
# [ $1 -lt 0 -o$1 -gt 30 ]
then
        echo "mdays is out of range"
        exit 2
fi

if (( $2 <= 20 ))       # [ $2 -lw 20 ]
then
        echo "size is out of range"
        exit 3
fi
```

```
# -xdev = do not search other partitiions
# -mtime =  n - # of days since file was modified
# -size = size of the fle in 512 -bytes blocks.
find / -xdev -mtime $1 -size +$2
SIESIEGFRIE@panther:~$ bigfiles
Usage: /home/siegfried/bin/bigfiles mdays size
SIEGFRIE@panther:~$ echo $?
1
SIEGFRIE@panther:~$ bigfiles 400 80
mdays is out of range
SIEGFRIE@panther:~$ echo $?
2
SIEGFRIE@panther:~$ bigfiles 25 2
size is out of range
SIEGFRIE@panther:~$ echo $?
SIEGFRIE@panther:~$
```

```
SIEGFRIE@panther:~$ bigfiles 2 25
find: `/lost+found': Permission denied
find: `/etc/chatscripts': Permission denied
find: `/etc/cups/ssl': Permission denied
… … …
/var/lib/apt/lists/us.archive.ubuntu.com_ubuntu_dist
s_precise-backports_multiverse_source_Sources
^C
SIEGFRIE@panther:~$ echo $?
130
SIEGFRIE@panther:~$
```

Checking For **`null`** Values

- When checking for **`null`** values in a variable, use double quotes to hold the **`null`** value or the **`test`** command will fail.

```
SIEGFRIE@panther:~$ cat bin/nulltest
#!/bin/bash
if [ "$name" = "" ] # [ ! "$name" ] or { -z "$name" ]
then
        echo The name variable is null
fi
SIEGFRIE@panther:~$ nulltest
The name variable is null
SIEGFRIE@panther:~$
```

Nested **`if`** Commands

- The if/else commands allows a two-way decision-making process.
- Format:

```
if   command
then
   command(s)
else
   command(s)
fi
```

bin/grepit

```
SIEGFRIE@panther:~$ cat bin/grepit
#!/bin/bash
# Script name: grepit
if grep "$1" /etc/passwd >& /dev/null; then
        echo "Found $1!"
else
        echo "Can't find $1."
fi
SIEGFRIE@panther:~$ grepit whoopsie
Found whoopsie!
SIEGFRIE@panther:~$ grepit Siegfried
Can't find Siegfried.
SIEGFRIE@panther:~$
```

bin/idcheck

```
SIEGFRIE@panther:~$ id
your user id is: 16131
You are not superuser.
SIEGFRIE@panther:~$ uid=16131(SIEGFRIE)
gid=100(users) groups=100(users)
SIEGFRIE@panther:~$idcheck
SIEGFRIE@panther:~$ cat bin/idcheck
#!/bin/bash
# Sciprname: idcheck
# Purpose: check user id to see if user is root.
# Only root has an uid of 0.
# Format for id output: uid=9496(ellie) gid=40
groups=40
# root's uid=0
```

```
id=`id | nawk -F'[=(]' '{print $2}'` #get user id
echo your user id is: $id
if (( id == 0 ))
then
        echo "You are superuser."
else
        echo "You are not superuser."
fi
SIEGFRIE@panther:~$
```

The **`if/elif/else`** Command

- The if/elif/else command allows multiway decision making.

if/elif/else Syntax

```
if command
then
   command(s)
elif command
then
   command
else
   command
fi
```

bin/tellme

```
SIEGFRIE@panther:~$ cat bin/tellme
#!/bin/bash
# Scriptname: tellme
# Using the ols-style test command

echo -n "How old are you? "
read age
if [ $age -lt 0 -o $age -gt 120 ]
then
        echo "Welcome to our planet! "
        exit 1
fi
```

```
if [ $age  -ge 0 -a $age -le 12 ]
then
        echo "A child is a garden of verses"

elif [ $age -gt 12 -a $age -le 19 ]
then
        echo " Rebel without a cause"
elif [ $age -gt 19 -a $age -le 29 ]
then
        echo "You got the world by the tail!!"
elif [ $age -gt 20 -a $age -le 39 ]
then
        echo "Thirty something..."
else
        echo "Sorry I asked"
fi
```

```
SIEGFRIE@panther:~$ tellme
How old are you? -1
Welcome to our planet!
SIEGFRIE@panther:~$ tellme
How old are you? 0
A child is a garden of verses
SIEGFRIE@panther:~$ tellme
How old are you? 14
 Rebel without a cause
SIEGFRIE@panther:~$ tellme
How old are you? 24
You got the world by the tail!!
SIEGFRIE@panther:~$ tellme
How old are you? 34
Thirty something...
SIEGFRIE@panther:~$
```

```
SIEGFRIE@panther:~$ tellme
How old are you? 44
Sorry I asked
SIEGFRIE@panther:~$
```

bin/tellme2

```
SIEGFRIE@panther:~$ cat bin/tellme2
#!/bin/bash
# Scriptname: tellme
# Using the new-style (( )) compound let command

echo -n "How old are you? "
read age
if (( $age < 0 || $age > 120 ))
then
        echo "Welcome to our plant! "
        exit 1
fi
```

```
if (( $age >= 0 && $age <= 12 ))
then
        echo "A child is a garden of verses"

elif (( $age > 12 && $age <= 19 ))
then
        echo " Rebel without a cause"
elif (( $age > 19 && $age <= 29 ))
then
        echo "You got the world by the tail!!"
elif (( $age > 20 && $age <39 ))
then
        echo "Thirty something..."
else
        echo "Sorry I asked"
fi
SIEGFRIE@panther:~$
```

File Testing

Test Operator	Test True if:
-b filename	Block special file
-c filename	Character special file
-d filename	Directory existence
-e filename	File existence
-f filename	Regular file existence and not a directory
-G filename	True if file exists and is owned nu the effective group id
-g filename	Set-group-ID is set
-k filename	Sticky bit is set
-L filename	File is a symbolic link

File Testing (continued)

Test Operator	Test True if:
-p filename	File is a named pipe
-O filename	File exists and is owned by the effective user ID
-r filename	file is readable
-S filename	file is a socket
-s filename	file is nonzero size
-t fd	True if fd (file descriptor) is opened on a terminal
-u filename	Set-user-id bit is set
-w filename	File is writable
-x filename	File is executable

Example – **bin/permcheck**

```
SIEGFRIE@panther:~$ cat bin/permcheck
#!/bin/bash
# Using the old-style test command [ ] single
brackets
# filename: permcheck
file=$HOME/fleas

if [ -d $file ]
then
        echo "$file is a directory"
elif [ -f $file ]
then
```

```
if [ -r $file -a -w $file -a -x $file ]
        then
                # nested if command
            echo "You have read, write and execute
permission on $file."
        fi
else
        echo "$file is neither a file nor a
directory."
fi
SIEGFRIE@panther:~$ permcheck
You have read, write and execute permission on
/home/siegfried/fleas.
SIEGFRIE@panther:~$
```

Example – **bin/permcheck2**

```
SIEGFRIE@panther:~$ cat bin/permcheck2
#!/bin/bash
# Using the new compound operator for test [[ ]]
# filename: permcheck2
file=$HOME

if [[ -d $file ]]
then
        echo "$file is a directory"
elif [[ -f $file ]]
then
```

```
      if [[ -r $file && -w file && -x $file ]]
      then    # nested if command
            echo "You have read, write and execute
permission on $file."
        fi
else
        echo "$file is neither a file nor a
directory."
fi
SIEGFRIE@panther:~$ permcheck2
/home/siegfried is a directory
SIEGFRIE@panther:~$
```

The `null` Command

- The null command is represented by a colon, is a built-in, do-nothing command that returns an exit status of 0.
- It is used as a placeholder after an if command when you have nothing to say, but you need a command or program to avoid producing an error message (you MUST have a command after a then statement).
- It can also be used to create an infinite loop.

Example - **bin/namegrep**

```
SIEGFRIE@panther:~$ cat bin/namegrep
#!/bin/bash
# filename: namegrep

name=$1
if grep "$name" $HOME/fleas >& /dev/null
then
        :
else
        echo "$1 not found in fleas"
        exit 1
fi
SIEGFRIE@panther:~$ namegrep fleas
SIEGFRIE@panther:~$ namegrep Seymour
Seymour not found in fleas
SIEGFRIE@panther:~$
```

null Command – An Example

```
SIEGFRIE@panther:~$ DATAFILE= # assigned null value
SIEGFRIE@panther:~$ : ${DATAFILE:=$HOME/c/hello.c}
# : does nothing := assigns a value to DATAFILE, it's set
permanently
SIEGFRIE@panther:~$ echo $DATAFILE
/home/siegfried/c/hello.c
SIEGFRIE@panther:~$ : ${DATAFILE:=$HOME/junk}
# Won't reset it
SIEGFRIE@panther:~$ echo $DATAFILE
/home/siegfried/c/hello.c
SIEGFRIE@panther:~$
```

Example – **bin/wholenum**

```
SIEGFRIE@panther:~$ cat bin/wholenum
#!/bin/bash
# Scriptname: wholenum
# Purpose: The expr command tests that the user
enters an integer

echo "Enter an integer"
read number
if expr "$number" + 0 >& /dev/null
then
        :
else
        echo "You did not enter an integer value."
        exit 1
fi
SIEGFRIE@panther:~$
```

```
SIEGFRIE@panther:~$ wholenum
Enter an integer
54
SIEGFRIE@panther:~$ wholenum
Enter an integer
76.7
You did not enter an integer value.
SIEGFRIE@panther:~$
```

The **`case`** Command

- The **`case`** command as a multiway branching command used as an alternative to the **`if/elif/else`** construct.
- The **`case`** variable's value is compared to **`value1`**, **`value2`**, etc. until a match is found.
- Case has a special case **`*`**, which indicates what is done when there is no match.

`case` Format

```
case variable
value1)
   command(s)
   ;;
value2)
   command(s)
   ;;
*)
   command(s)
   ;;
esac
```

`case` – An Example

```
SIEGFRIE@panther:~$ cat bin/xcolors
#!/bin/bash
# Scriptname: xcolors

echo -n "Choose a foreground color for your xterm
window:"
read color

case "$color" in
[Bb]l??)
        xterm -fg blue -fn terminal &
        ;;
[Gg]ree*)
        xterm -fg darkgreen -fn terminal &
        ;;
```

```
red | orange)   # | means "or"
        xterm -fg "$color" -fn terminal &
        ;;
*)
        xterm -fn terminal
        ;;
esac
echo "Out of case command"

SIEGFRIE@panther:~$
```

bin/setmenu

```
SIEGFRIE@panther:~$ cat bin/setmenu
echo "Select a terminal setting"
cat <<- ENDIT
   1) unix
   2) xterm
   3) sun
ENDIT
read choice
case "$choice" in
1)
        echo "We chose UNIX(tm)"
        ;;
2)
        echo "We chose xterm"
        ;;
```

```
3)
        echo "We chose sun"
        ;;
*)
        echo "We chose something else."
        ;;
esac
SIEGFRIE@panther:~$ setmenu
Select a terminal setting
   1) unix
   2) xterm
   3) sun
1
We chose UNIX(tm)
SIEGFRIE@panther:~$
```

```
SIEGFRIE@panther:~$ setmenu
Select a terminal setting
   1) unix
   2) xterm
   3) sun
2
We chose xterm
SIEGFRIE@panther:~$ setmenu
Select a terminal setting
   1) unix
   2) xterm
   3) sun
4
We chose something else.
SIEGFRIE@panther:~$
```

Looping in Bash – The **`for`** Command

- The for command is used to execute commands a finite number of times on a list of items.
- The format is

```
for variable in word_list
do
    command(s)
done
```

- **`variable`** will take on the value of each of the words in the list.

bin/forloop

```
SIEGFRIE@panther:~$ cat bin/forloop
#!/bin/bash
# Scriptname: forloop
for pal in Tom Dick Harry
do
        echo "Hi $pal"
done
echo "Out of loop"
SIEGFRIE@panther:~$ forloop
Hi Tom
Hi Dick
Hi Harry
Out of loop
SIEGFRIE@panther:~$
```

bin/mailer

```
SIEGFRIE@panther:~$ cat mylist
tom
patty
ann
jake
SIEGFRIE@panther:~$ cat bin/mailer
#!/bin/bash
# Scriptname: mailer
# `cat my list` command substitute the alternate way
for person in $(cat mylist)
do
        cat letter | sed "s/myFriend/$person/g"
        echo "Personalized the letter for $person"
done
SIEGFRIE@panther:~$
```

```
SIEGFRIE@panther:~$ mailer
Dear tom,

  It is a pleasure to know you.

RMS
Personalized the letter for tom
Dear patty,

  It is a pleasure to know you.

RMS
Personalized the letter for patty
```

```
Dear ann,

  It is a pleasure to know you.

RMS
Personalized the letter for ann
Dear jake,

  It is a pleasure to know you.

RMS
Personalized the letter for jake
SIEGFRIE@panther:~$
```

bin/backup

```
SIEGFRIE@panther:~/junk$ pwd
/home/siegfried/junk
SIEGFRIE@panther:~/junk$ cat $HOME/bin/backup
#!/bin/bash
# Scriptname: backup
# Purpose: Create backup files and store
# them in a backup directory
#

dir=$HOME/junk/backupstuff
ls $HOME/junk
```

```
for file in `ls $HOME/junk`
do
        if [ -f $file ]
        then
                cp $file $dir/$file.bak
                echo "$file is backed up in $dir"
        fi
done
SIEGFRIE@panther:~/junk$ backup
411  backupstuff  cookie  file1  file2  oreo  phone-
book  temp  time.out  xyzzy
411 is backed up in /home/siegfried/junk/backupstuff
cookie is backed up in
/home/siegfried/junk/backupstuff
file1 is backed up in
/home/siegfried/junk/backupstuff
```

```
file2 is backed up in
/home/siegfried/junk/backupstuff
oreo is backed up in
/home/siegfried/junk/backupstuff
phone-book is backed up in
/home/siegfried/junk/backupstuff
temp is backed up in
/home/siegfried/junk/backupstuff
time.out is backed up in
/home/siegfried/junk/backupstuff
xyzzy is backed up in
/home/siegfried/junk/backupstuff
SIEGFRIE@panther:~/junk$
```

$* and $@

- `$*` and `$@` can be used as part of the list in a for loop or can be used as par of it.
- When expanded `$@` and `$*` are the same unless enclosed in double quotes.
  - `$*` is evaluated to a single string while `$@` is evaluated to a list of separate word.

bin/greet

```
SIEGFRIE@panther:~$ cat bin/greet
#!/bin/bash
# Scriptname: greet
for name in $*
do
        echo Hi $name
done
SIEGFRIE@panther:~$ greet Bob Carol Ted Alice
Hi Bob
Hi Carol
Hi Ted
Hi Alice
SIEGFRIE@panther:~$
```

bin/permx

```
SIEGFRIE@panther:~/bin$ cat permx
#!/bin/bash
# Scriptname: permx

for file        # Empty wordlist
do
        if [[ -f $file && ! -x $file ]]
        then
                chmod +x $file
                echo $file now has execute
permission
        fi
done
SIEGFRIE@panther:~/bin$
```

```
SIEGFRIE@panther:~/bin$ permx alloc ampersand
alloc now has execute permission
ampersand now has execute permission
SIEGFRIE@panther:~/bin$ ls -l a*
-rwxr-xr-x 1 SIEGFRIE users 5047 Jul 29  2009 alloc
-rwxr-xr-x 1 SIEGFRIE users 4843 Nov 19  2012
ampersand
-rwxr-xr-x 1 SIEGFRIE users  526 Oct 25 11:41 arg2
-rwxr-xr-x 1 SIEGFRIE users  517 Oct 22 19:13 args
-rwxr-xr-x 1 SIEGFRIE users 8379 Oct 11 11:05 atest
-rwxr-xr-x 1 SIEGFRIE users 5011 Jul 13  2009 atoi
-rwxr-xr-x 1 SIEGFRIE users 5600 Oct 10  2012 atoif
-rwxr-xr-x 1 SIEGFRIE users 4770 Jun 23  2009
autoincr
-rwxr-xr-x 1 SIEGFRIE users 5078 Jun 23  2009 avg
SIEGFRIE@panther:~/bin$
```

while Command

- The while command evaluates the command following it and, if its exit status is 0, the commands in the body of the loop are exeeceuted.
- The loop continues until the exit status is nonzero.
- Format:

```
while command
do
   command(s)
done
```

bin/numm

```
SIEGFRIE@panther:~$ cat bin/numm
#!/bin/bash
# Scriptname: numm
num=0
while (( $num < 10 ))
do
        echo -n "$num"
        let num+=1
done
echo -e "\nAfter loop exits, continue running here"
SIEGFRIE@panther:~$ numm
0123456789
After loop exits, continue running here
SIEGFRIE@panther:~$
```

bin/quiz

```
SIEGFRIE@panther:~$ cat bin/quiz
#!/bin/bash
# Scriptname: quiz

echo "Who was the 2nd U. S. president to be
impeached?"
read answer
while [[ "$answer" != "Bill Clinton" ]]
do
        echo "Wrong..try again."
        read answer
done
echo "You got it!"
SIEGFRIE@panther:~$
```

```
SIEGFRIE@panther:~$ quiz
Who was the 2nd U. S. president to be impeached?
George Washington
Wrong..try again.
Abe Lincoln
Wrong..try again.
Bill Clinton
You got it!
SIEGFRIE@panther:~$
```

The **`until`** Command

- until works like the while command, except it execute the loop if the exit status is nonzero (i.e., the command failed).
- Format:

```
until command
do
   command(s)
done
```

bin/hour

```
SIEGFRIE@panther:~$ cat bin/hour
#!/bin/bash
# Scriptname: hour
hour=0
until (( hour > 24 ))
do
        case    "$hour" in
        [0-9]|1[0-1]) echo "Good morning!"
                ;;
        12) echo "lunch time."
                ;;
        1[3-7]) echo "Siesta time."
                ;;
        *) echo "Good night."
                ;;
        esac
```

```
        (( hour +=1 ))  # Don't forget to increment
                        # the hour
done
SIEGFRIE@panther:~$ hour
Good morning!
… …
Good morning!
lunch time.
Siesta time.
… …
Siesta time.
Good night.
…
Good night.
SIEGFRIE@panther:~$
```

The **select** Command

- The select command allows the user to create menus in **bash**.
- A menu of numerically listed items is displayed to **stderr**, with **PS3** used to promp the user for input.
- Format:

```
select var in wordlist
do
   command(s)
done
```

bin/runit

```
SIEGFRIE@panther:~$ cat bin/runit
#!/bin/bash
# Scriptname: runit

PS3="Select a progam to execute: "
select program in 'ls -F' pwd date
do
        $program
done
SIEGFRIE@panther:~$ runit
1) ls -F
2) pwd
3) date
Select a program to execute
```

```
Select a program to execute: 1
160L2Handout.pdf    data.dat     java/      sample.c
… … …
data                    j        recipes/
Select a program to execute: 2
/home/siegfried
Select a program to execute: 4
Select a program to execute: 3
Sun Nov  3 10:55:36 EST 2013
Select a program to execute: 4
Select a program to execute: 5
Select a program to execute: 6
Select a program to execute: ^C
SIEGFRIE@panther:~$
```

Commands Used With **`select`**

- **`select`** will automatically repeat and has do mechanism of its own to terminate. For this reason, the **`exit`** command is used to terminate.
- We use **`break`** to force an immediate exit from a loop (but not the program).
- We use **`shift`** to shift the parameter list one or more places to the left, removing the displaced parameters.

bin/goodboys

```
SIEGFRIE@panther:~$ cat bin/goodboys
##!/bin/bash
# Scriptname: goodboys

PS3="Please choose one of the three boys : "
select choice in tom dan guy
do
        case "$choice" in
        tom)
           echo Tom is a cool dude!
           break;; #break ouf of the select loop
        dan | guy )
           echo "Dan and Guy are both wonderful."
           break;;
```

```
      *)
       echo "$REPLY is not one of your choices."1>&2
       echo "Try again."
            ;;
      esac
done
SIEGFRIE@panther:~$ goodboys
1) tom
2) dan
3) guy
Please choose one of the three boys : 4
4 is not one of your choices.
Try again.
Please choose one of the three boys : 5
5 is not one of your choices.
Try again.
Please choose one of the three boys :
```

```
Please choose one of the three boys : 6
6 is not one of your choices.
Try again.
Please choose one of the three boys : 1
Tom is a cool dude!
SIEGFRIE@panther:~$ goodboys
1) tom
2) dan
3) guy
Please choose one of the three boys : 2
Dan and Guy are both wonderful.
SIEGFRIE@panther:~$ goodboys
1) tom
2) dan
3) guy
Please choose one of the three boys :
```

```
Please choose one of the three boys : 3
Dan and Guy are both wonderful.
SIEGFRIE@panther:~$
```

bin/ttype

```
SIEGFRIE@panther:~$ cat bin/ttype
#!/bin/bash
# Scriptname: ttype
# Purpose: set the terminal type
COLUMNS=60
LINES=1  # forces menu items to be printed on one
line
PS3="Please enter the terminal type: "
select choice in wuse50 vt200 xterm sun
do
        case "$REPLY" in
        1)
             export TERM=$choice
             echo "TERM is $choice"
             break;; # break out of the select loop
```

```
        2 | 3)
                export TERM=$choice
                echo "TERM=$choice"
                break;;
        4)
                export TERM=$choice
                echo "TERM=$choice"
                break;;
        *)
                echo -e \
"$REPLY is not a valid choice.  Try again\n" 1>&2
                REPLY= # Causes the menu to be
                       # redisplayed
                ;;
        esac
done
SIEGFRIE@panther:~$
```

```
SIEGFRIE@panther:~$ ttype
1) wuse50
2) vt200
3) xterm
4) sun
Please enter the terminal type: 4
TERM=sun
SIEGFRIE@panther:~$ ttype
1) wuse50
2) vt200
3) xterm
4) sun
Please enter the terminal type:
TERM=xterm
SIEGFRIE@panther:~$
```

bin/shifter

```
SIEGFRIE@panther:~$ cat bin/shifter
#1/bin/bash
# Scriptname: shifter
set joe mary tom sam
shift
echo $*
set $(date)
echo $*
shift 5
echo $*
shift 2
SIEGFRIE@panther:~$ shifter
mary tom sam
Sun Nov 3 13:49:11 EST 2013
2013
SIEGFRIE@panther:~$
```

bin/dater

```
SIEGFRIE@panther:~$ cat bin/dater
#!/bin/bash
# Scriptname: dater
# Purpose: set positional parameters with the set
# command and shift through the parameters
set $(date)
while (( $# > 0 ))
do
        echo $1
        shift
done
SIEGFRIE@panther:~$
```

```
SIEGFRIE@panther:~$ dater
Sun
Nov
3
13:53:41
EST
2013
SIEGFRIE@panther:~$
```

bin/loopbreak

```
SIEGFRIE@panther:~$ cat bin/loopbreak
#!/bin/bash
# Scriptname: loopbreak
while true; do
        echo Are you ready to move on\?
        read answer
        if [[ "$answer" == [Yy] ]]
        then
                break
        else
                echo whatever we want to do
        fi
done
echo "Here we are."
SIEGFRIE@panther:~$
```

```
SIEGFRIE@panther:~$ loopbreak
Are you ready to move on?
sdaflfsda
whatever we want to do
Are you ready to move on?
no
whatever we want to do
Are you ready to move on?
maybe
whatever we want to do
Are you ready to move on?
never!!
whatever we want to do
Are you ready to move on?
y
Here we are.
SIEGFRIE@panther:~$
```

bin/months

```
SIEGFRIE@panther:~$ cat bin/months
#!/bin/bash
# Scriptname: months
for month in Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov
Dec
do
        for week in 1 2 3 4
        do
          echo -n "Processing the month of $month.  OK?"
          read ans
          if [ "$ans" = n -o -z "$ans" ]
          then
            continue 2
          else
            echo -n "Process week $week of $month?"
            read ans
```

```
          if [ "$ans" = n -o -z "$ans" ]
          then
            continue
          else
            echo "Now processing week $week of $month."
            sleep 1
            echo "Done processing..."
          fi
        fi
        done
done
SIEGFRIE@panther:~$ months
Processing the month of Jan.  OK?y
Process week 1 of Jan?1
Now processing week 1 of Jan.
Done processing...
Processing the month of Jan. OK?
```

```
Processing the month of Jan.  OK?y
Process week 2 of Jan?y
Now processing week 2 of Jan.
Done processing...
Processing the month of Jan.  OK?n
Processing the month of Feb.  OK?n
Processing the month of Mar.  OK?n
… … …
Processing the month of Sep.  OK?y
Process week 1 of Sep?n
Processing the month of Sep.  OK?y
Process week 2 of Sep?2
Now processing week 2 of Sep.
Done processing...
Processing the month of Sep.  OK?n
… … …
Processing the month of Dec.  OK?n
SIEGFRIE@panther:~$
```

bin/numberit

```
SIEGFRIE@panther:~$ cat bin/numberit
#!/bin/bash
# Progra name: numberit
# Put line numbers on all lines of memo
if (( $# < 1 ))
then
        echo "Usage: $0 filename" ?&2
        exit 1
fi
count=1 #Initialize count
cat $1 | while read line
# Input is cming from file provded at command line
do
      (( count == 1 )) && echo "Processing $1..." >
/dev/tty
```

```
        echo -e "$count\t$line"
        let count+=1
done > tmp$$
mv tmp$$ $1
SIEGFRIE@panther:~$ cat memo2
abc
def
ghi
SIEGFRIE@panther:~$ numberit memo2
Processing memo2...
SIEGFRIE@panther:~$ cat memo2
1       abc
2       def
3       ghi
SIEGFRIE@panther:~$
```