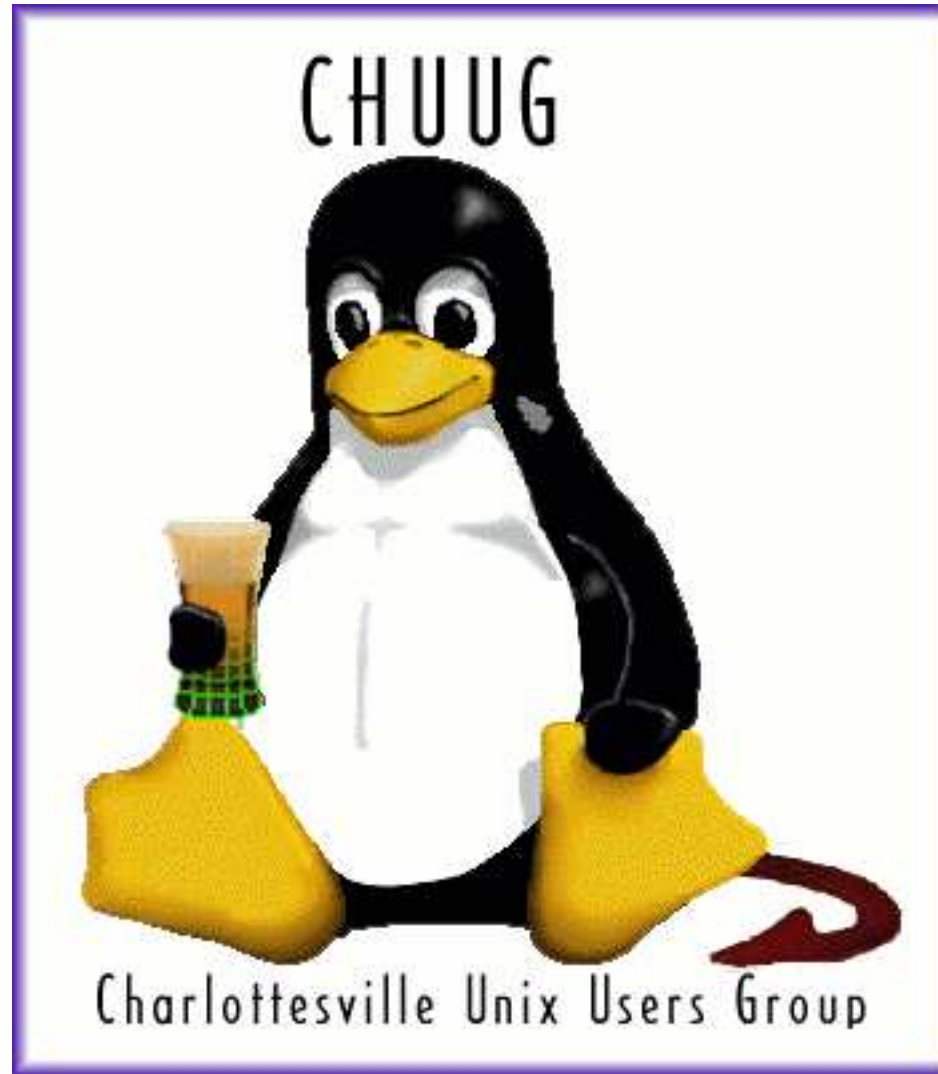


Advanced Bash Scripting



Joshua Malone (jmalone@ubergeeks.com)

Why script in bash?

- You're probably already using it
- Great at managing external programs
- Powerful scripting language
- Portable and version-stable
- Almost universally installed

Basic syntax: statements and line format

- Start the script with `#!/path/to/bash` like most scripts
 - Beware: `/path/to/bash` differs between unices
 - ...and even distros of Linux **sigh**

- No semicolon at the end of a line
- Semicolons can separate multiple statements on the same line
- Most statements are either external programs or bash "builtins"
 - See `man builtins`
- No parenthesis around function arguments

Basic syntax: variables

Variable assignment

FOO=BAR No spaces!

Variable expansion

\$FOO

\${FOO} Safer way -- brace protected

Basic syntax: conditional evaluation

IF statement

```
if <command>           Command is any process that exists true or false
then
    <commands>
else
    <commands>
fi
```

You can use a semicolon to put more than one part of this structure on one line

```
if <command> ; then
```

Basic syntax: conditional evaluation

CASE statement

```
case <variable> in
    <condition1> )
        <commands>
        ; ;
    <condition2> )
        <commands>
        ; ;
    * )
        <commands>
        ; ;
esac
```

Default match

<condition> statements are matched according to globbing rules (more later)

More bash syntax

While loop

```
while <command>
do
    <commands>
done
```

For loop

```
for variable in <list>
do
    <commands>
done
```

- List is an IFS-separated list of literals or a variable containing one
- IFS is the "inter-field separator" -- we'll get to this later (usually a space)

External programs

It's bash - just type the command :)

Capturing output of a command

`FOO=`prog`` Backticks are more portable

`FOO=$(prog)` But parenthesis are easier to read, safer and also nest (more later)

Sending output to a command

`echo $FOO | prog`

Combining the two

`FOO=$(echo $BAR | prog)`

Background programs are post-fixed with an `&` just like normal

The special variable `#!` holds the PID of the last background task started

User output

Display output using `echo` builtin or an external program like `printf`

`echo "foo"` Outputs "foo" with trailing newline

`echo -n "foo"` Outputs "foo" but doesn't send a newline

Escape sequences are parsed if the `-e` option to `echo` is given

`echo -e "\tfoo"` Outputs "foo" with a tab character in front and a trailing newline

User input

Read input from user using `read`

`read foo` Accepts user input and stores it into variable `foo`

`read -p "<string>" foo` Displays the prompt `<string>` and reads user input into `foo`

`read -t 30 foo` Read input into `foo` but time out after 30 seconds

`read -s foo` Read input into `foo` but don't echo it to the terminal

Tests

Remember that `if` just tests the return value (true/false) of a command.

All tests are implemented in external binaries, especially the `test` or `[` program

Types of tests

- string (`-z`, `=`, `!=`, ...)
- integer (`-eq`, `-gt`, `-lt`, ...)
- file (`-f`, `-d`, `-w`, ...)

Basic Syntax

```
if [ $FOO = $BAR ]
```

```
if [ $count -lt 5 ]
```

See `man test` for more tests.

Basic math in bash

Bash has basic built-in INTEGER math evaluation using `$((<expression>))`

Examples:

```
echo $(( 4 + 5 ))          -> "9"
```

```
FOO=4
```

```
echo $(( $FOO + 5 ))      -> "9"
```

```
BAR=$(( 10 / 4 ))
```

```
echo $BAR                 -> "2"    Remember - integer math
```

For more complex math, or floating point, you'll need to use an external calculator like `bc`.

Command line arguments to scripts

The special variables `$1`, `$2`, etc., hold the arguments given on the command line

`$0` the name of the script as executed by the shell

`$#` the number of arguments passed to the script

`$*` is an IFS-separated list of all command line arguments

`$@` is a list of all command line arguments individually double-quoted

The built-in command `shift` moves the CLA's down (to the left) one and discards `$1`
(`$2` becomes `$1`, `$3` becomes `$2`, etc.)

This can be used to iterate over the list or handle optional arguments

The external program `getopt` is also useful for processing a large number of arguments

Functions in bash

- Declare function by placing parenthesis after the function name
- Place function commands inside curly braces

```
function function_name () {  
    <commands>  
}
```

The keyword "function" is not necessary but improves readability

Arguments to bash functions are accessed just like CLAs using `$1`, `$2`, etc.

Calling bash functions

To call a function, type it's name like any other command

Arguments to bash functions are not put inside parenthesis

```
function foo () {  
    echo "Argument 1 is $1"  
}
```

```
foo bar    -> outputs "Argument 1 is bar"
```

Shell globbing

Bash shell performs character matching against special symbols

- process called "globbing"

*	Any character or characters
?	Any single character
[abc]	Any 1 of the characters a, b, or c
[^abc]	Any 1 character other than a, b, or c
{ a* , b* }	Any of the patterns enclosed in braces (matches a* or b*)

- Invoke bash with `-f` flag to disable globbing

Breather

Okay - that was your 15 minute crash course in bash.

Everybody with me?

Good - lets get to the fun stuff :)

Advanced variable expansion

Other ways to evaluate a variable

`${#foo}` Number of characters in (length of) foo

`${foo:3:5}` Characters 3 through 5 of foo

`${foo:4}` Foo beginning from the fourth character (chars 4 through end)

`${foo#STRING}` Foo, but with the shortest match of "STRING" removed from the beginning

`${foo%STRING}` Foo, but with the shortest match of "STRING" removed from the end

`${foo%%STRING}` Foo, but with largest match of "STRING" removed from the end

`${foo##STRING}` Foo, but with largest match of "STRING" removed from the beginning

Advanced variable expansion (cont.)

`${foo/bar/baz}` Foo, but with first occurrence of string "bar" replaced by string "baz"

`${foo//bar/baz}` Foo, but with all occurrences of string "bar" replaced by string "baz"

Test shortcuts

You can use the "logical and" operator `&&` as a short "if" statement

```
if [ $1 -eq 0 ]  
then  
    <do stuff>  
fi
```

Is equivalent to

```
[ $1 -eq 0 ] && <do stuff>
```

Dealing with unset variables

If a variable hasn't been set to a value, expanding it results in a NULL

- This is not an error condition!

Providing default values for unset variables:

`${foo:-bar}` If foo is unset, substitute the value "bar" of instead

`${foo:-$bar}` If foo is unset, substitute the value of variable bar instead

`${foo:=bar}` If foo is unset, substitute the value bar and set foo=bar

The "eval" command

The eval command constructs a statement and then evaluates it

- Can be used to get variable-names in bash

Example: set variable FOO to last argument passed to script

```
eval "FOO=\${$#}"
```

Remember \$# is the number of arguments passed to the script

Example: if I run a script with 4 arguments

- First: string "FOO=\$4" is constructed
- Second: FOO is set to value of \$4

Manipulating the IFS

IFS is the inter-field separation character

- Default IFS is a space (" ")
- IFS is set like any other variable

Example: parsing /etc/passwd

```
line=$(grep $name /etc/passwd)    # assuming name already set
OLDIFS="$IFS"                     # always back up IFS before changing
IFS=:
x=0
for i in $line; do
    eval "field${x}=\"\${i}\""
    x=$(( x+1 ))
done
IFS="$OLDIFS"

echo "Shell for $name is $field6"
```

Storing functions in a different file

Bash can load in the contents of an external file using `source` command

Source command is abbreviated `'.'`

Example:

```
. ~/shell-library.sh
```

WARNING: if the sourced file is absent your script will abort

Protect it with a file test:

```
[ -f $library ] && . $library
```

Here documents

You can feed a long block of text into a command or variable using a "Here document"

Example: function to print out a help message

```
function print_help() {  
    cat << EOF  
    Usage: program [-f] <input> <output>  
        -f:  some flag  
        input:  input file in some format  
        output: output file in some format  
    EOF  
}
```

The string "EOF" can be any string NOT included in the contents of your Here document.

More Here documents

You can feed the contents of a Here doc to any program that accepts input via stdin

Example: applying edits to a config file

```
ex - /etc/ssh_config << EOF
/# Host */s/# //
/# ForwardX11 no/s/#/ /
s/X11 no/X11 yes/
a
    ForwardX11Trusted yes
.
x
EOF
```

This script sends input to the editor ex (vi in colon-mode)

- Uncomments the default host stanza
- Enables X11 forwarding
- Adds X11Trusted forwarding after the ForwardX11 line

Result is similar to applying a patch but more resistant to changes in the default file

Curly braces and redirection

You can redirect the output of a whole block of statements using curly braces

Example: Editing a MOTD with a standard first line

```
"System message for Tue Apr 24 19:01:43 EDT 2007"
```

```
...
```

```
{
```

```
    echo '/System message for /d'
```

```
    echo 'a'
```

```
    echo -n 'System message for '
```

```
    date
```

```
    echo ''
```

```
    echo '.'
```

```
    echo 'x'
```

```
    ....
```

```
} | ex - /etc/motd
```

Let's finish off with a few examples

Example: Advanced xinitrc startup

Using the "wait" command, we can start desk accessories after starting the window manager

```
eval $(ssh-agent)
```

```
xmodmap ~/.xmodmap-winkey
```

```
if [ -x "$(which xclock)" ]; then
```

```
    xclock &
```

```
fi
```

```
wmaker &                This launches the window manager
```

```
WMPID=$!
```

```
ssh-add ~/.ssh/id_dsa    The window manager is running already so it can  
                          manage the ssh-askpass window
```

```
wait ${WMPID}           This command simply blocks until the PID given exits
```

```
ssh-add -k              After the "wait", the windowmanager has exited and we can clean up
```

Example: Writing a log file

For complex scripts, I like to write a log file that's separate from stdout's user interaction

Using the standard output redirection

```
function initlog () {
    LOGFILE=$1
    echo '' > ${LOGFILE}
}
function log () {
    echo $* >> ${LOGFILE}
}

initlog "script.log"
log Starting process foo
```

Example: Running a log window

We can expand our logging example by opening a window to show the log to the user

```
initlog "script.log"
```

```
xterm -e "tail -f ${LOGFILE}" &  
LOGWIN=$!
```

```
log Some messages
```

```
# When the script is finished  
kill $LOGWIN
```

Example: Re-creating useful utils in bash

BSD has this great little utility called 'jot' which can print a sequence of numbers

```
[user@host ~]# jot 5 10  
10  
11  
12  
13  
14
```

This is especially useful for creating the list needed for a 'for' loop

Since I've never seen this for any Linux distros, I decided to just re-create it in bash.

My version will just print the numbers between \$1 and \$2 - good enough for me

Also be nice if it can zero-pad the numbers

Example: BSD jot in bash

```
function usage () {  
cat << EOF  
Usage: $0 [-p length] <start> <end>
```

Generates a series of numbers from start to end in
interger steps.

```
-p <n>: pad smaller number out to n digits
```

```
EOF
```

```
exit 1
```

```
}
```

Example: BSD jot in bash (cont')

```
padlen=0
if [ "$1" = '-p' ]; then
    padlen="$2"
    shift; shift
fi
[ -z "$2" ] && usage
begin=$1
end=$2
x=$begin
while [ $x -le $end ]; do
    number=$x
    if [ $padlen -gt 1 ]; then
        while [ ${#number} -lt $padlen ]; do number="0${number}"; done
    fi
    echo -n "$number"
    [ $x -lt $end ] && echo -n " "
    x=$(( $x + 1 ))
done
```