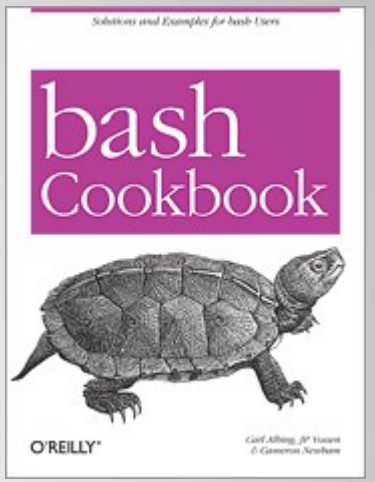


bash
tutorial

- Carl Albing, JP Vossen
- www.bashcookbook.com

Lots to do...

- do we still need command line?
- rapid fly-by of bash
    - as command interpreter
    - as language
- a few power-user features

Who are we here today?

- S/W Developer
- Sys Admin
- Hobbyist
- Professional
- ???

Command Line

- Isn't that so “last century”?
- Try doing this with a GUI:
  - rename 280 files
  - move only files with “Tea”
  - copy a file every 10 minutes
  - tweak the content of 50 files
  - add a feature (e.g. sort)
- Not just for system admins
  - users, developers, too
  - general “housekeeping”
- Even available for Windows!

bash – the big picture

- a breakthrough – separate from OS
- commands as piece-wise tasks
  - small, single or simple
- connect-able commands
  - like Legos®
  - I/O redirection

bash - commands

- simple use:
- *prompt command arguments*
- `$ ls`
- `...`
- `$ echo hi`
- `hi`
- `$`

bash - connections

- I/O redirection
  - command > output

```
ls > mydir.txt
```

  - command < input

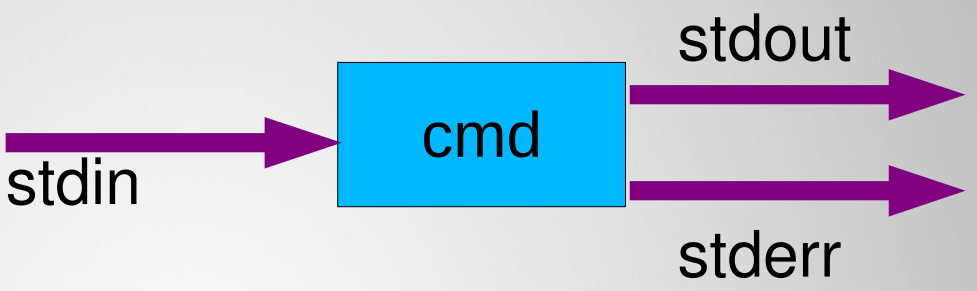
```
wc < mydata
```

  - command1 | command2

```
ls | wc
```

- avoids feature creep
- relies on streams not GUIs

bash - connections



- stdout and stderr

```
cmd > outfile  2> errfile
cmd > logfile  2>&1
cmd 2>&1  | next command
cmd |& next command   # 4.x
```

- expert challenge:
  can you connect stderr to a pipe?

more redirections

```
$ set noclobber
$ cmd > file
$ cmd > file      # errors this time
$ cmd >| file     # overwrites


$ cmd > file
$ cmd >> file
```

more redirections

```
$ # equivalent
$ cmd &> file
$ cmd >& file
$ cmd > file 2>&1
```

“here” redirections

```
$ cmd <<EOF
$ cmd <<'EOF'
$ cmd <<\EOF
$ cmd <<-EOF
$ cmd <<anytxt

$ cmd <<<string
```

closing connections

- as part of redirection
- important for daemon scripts
- use `&-`

```
bgcmd >/dev/null 2>&1 <&-   &

bgcmd &>/dev/null <&-   &
```

bash connections

- “process substitution”
- using `<(...)` and `>(...)`

```
$ wc <(ls | grep -e '\.c')
       5       5      46 /dev/fd/63
$
```

bash beyond commands

- programming language
- basic operation is invocation
- variables
  - integers
  - strings
  - arrays
- control structures
  - branching
  - looping

bash variables

```
name=value
$name
${name}

name=value command
    HOME=/tmp scriptfile


echo $VAR
printf “count is %d\n” $CNT
```

- builtin, efficient, portable

variable manipulations

```
${name:-default}
${name:=default}
${name:+altvalue}
${name:?errmsg}
```

bash – string manipulations

- removing leading text

```
${name#pre}
${name##pre}
```

bash – string manipulations

- remove trailing text

```
${name%post}
${name%%post}
```

bash – string manipulations

- substitute

```
${name/pattern/string}
${name//pattern/string}
```

string manipulations

- substring:

```
${name:offset}
${name:offset:length}
${name: -offset:length}
${name:offset:-length}
```

- string length:

```
${#name}
```

string manipulation: desperation

```
while read ALINE
do
  for ((i=0; i<${#ALINE}; i++))
  do
    ACHAR=${ALINE:i:1}
    ...
  done
done
```

examples

examples

```
for FN in *.JPG
do
  mv “$FN” “${FN/JPG/jpg}”
# or:
  mv “$FN” “${FN%JPG}jpg”
done
```

- which is safer? do you know why?

bash – numerical

`let` statement

```
$(( ... ))
```

```
let CNT++
$((CNT++))
```

```
let BSUM+=(AVAL/PER)
$(( BSUM += ( AVAL / PER ) ))
```

- ***integer only! integer division!***
- no spacing with `let`

“`if`” statement

```
if list ; then cmds ; elif blist ; else cmds ;
fi
```

- list can be just a program or a whole pipeline
  - return 0 ==> “true”
- one such program is `test`
  - also known as “`[`”
  - a shell builtin

```
if [ $VAR -gt $LIMIT ] ; then ...
if [ $VAR ] ; then ...
```

"`if`" – newer syntax

- bash also supports `[[ ... ]]`
  - newer syntax
  - additional comparisons
    - `a == b` , `a != b` where b can be a shell pattern
    - `a =~ b` , where b can be a *regular expression*

“`if`” – newer syntax

- bash supports numerical expressions `((...))`
  - newer syntax
  - no need to use `$` in `$vars` (except for `$1`...)
  - more “natural” numerical comparisons, expr.

using no “if”

```
cd /tmp/myplace && echo got it
[ $VAR ] && rm $VAR
test $VAR && cat $VAR
$0 stop  &&  $0 start

test -x $SNMPD || exit 5
```

compound commands

```
(list)
{ list ; }
((expression))
[[ expression ]]
```

- all will work with “if” statement
- spaces, semi required for {...}
- ((...)) is arithmetic, (...) is subshell
- list is commands
- expression is not

for loop

```
for name [ in word ] ; do list ; done
```

- examples:

```
for fn ; do echo $fn; done >
   file.out

for fn in *.c; do cc -c $fn; done
```

```
for (( expr1 ; expr2 ; expr3 )) ; do
   list; done
```

- example:

```
for ((i=0; i<10; i++)); do echo $i;
   done
```

other control structures

```
while list; do list; done
until list; do list; done
case word in; pattern) list ;; esac
select var in word ; do list ; done
```

example

```
#!/bin/bash
# cookbook filename: dbiniter
# initialize databases from a standard file
# creating databases as needed.
DBLIST=$(mysql -e "SHOW DATABASES;" | tail +2)
select DB in $DBLIST "new..."
do
    if [[ $DB == "new..." ]]
    then
        printf "%b" "name for new db: "
        read DB rest
        echo creating new database $DB
        mysql -e "CREATE DATABASE IF NOT EXISTS $DB;"
    fi

    if [ "$DB" ]
    then
        echo Initializing database: $DB
        mysql $DB < ourInit.sql
    fi
done
```

bash – arrays

- one dimensional
- zero-based
- example uses:

```
declare -a name
name[5]=”blah”
name=(orange green yellow)
echo ${name[5]}
${#name[*]}
${#name[5]}
```

Associative Arrays

- As of bash 4.x
- String as index to an array
- Let computer do the hashing/lookup
- Available in other languages
  - python, perl, awk, others
- Extends bash array syntax

(Assoc.) Arrays - syntax

```
declare -a
$VAR[2]
$VAR[$NUM]

declare -A
$VAR[“indx”]
$VAR[$NDX]
```

```
# using assoc. arrays
 declare -A COUNTER
 while read app lang size
 do
   let COUNTER[$lang]++
 done
```

(Assoc.) Arrays - syntax

`${#COUNTER[*]}`

size of the array

`${!COUNTER[@]}`

list of elements of the array

```
# using assoc. arrays
for INDX in "${!COUNTER[@]}"
do
 printf “%s %d\n” $INDX $COUNTER[$INDX]
done
```

(Assoc.) Arrays - example

```
$ cat aa.data
mypile python 100
hardwork C 10284
fstdev java 4118
lakfjd C 413
aljjll java 870
zxcv perl 13432
ajkl0q C 13478
x.mx C 908
$
$ ./aaex.sh <aa.data
4 apps use C
2 apps use java
1 apps use python
1 apps use perl
$
```

(Assoc.) Arrays - example

```
# use associative arrays to count occurances
#
declare -A COUNTER
# first time, the array entry is an empty string
while read app lang size
do
        let COUNTER[$lang]++
done
#
# no order guaranteed for ${!...[*]}
# so we pipe into sort
for INDX in "${!COUNTER[@]}"
do
# NB the $ in "$INDX" is crucial to make a
# valid index value in the array reference
    printf "%d apps use %s\n" \
            ${COUNTER[$INDX]} $INDX
done | sort -rn
```

function definition

```
function plural ()
{
    if [ $2 -eq 1 -o $2 -eq -1 ]
    then
        echo ${1}
    else
        echo ${1}s
    fi
}
```

- not just { } but any compound statement
- add redirection, e.g. usage function

calling the function

- like any shell command
  - no parentheses on arguments

```
while read num name
do
    printf “%s “ $num
    plural "$name" $num
done
```

command substitution

- run a command
- capture the output
- newlines in output become spaces

  `$(` *cmd* `)`
- as in:

```
VAR=$(ls)
for FN in $(ls) ; do ...; done
```

Do you know...

- The command substitution

  `$(cat file)`

  - can be replaced by the equivalent but faster

    `$(< file)`

Do you know...

- how to:
  - redirect in/out of entire loops, functions
  - parse/set new args

```
function usage ()
{
...
} >&2


set -- $foo $bar
```

Connect stderr to |

- swapping stderr and stdout

```
myscript 3>&1 1>&2 2>&3
```

- to pipe stderr to next command
  - remember special order on a pipe

```
scrpt 3>&1 1>log.out 2>&3- | acmd
```

Net redirection

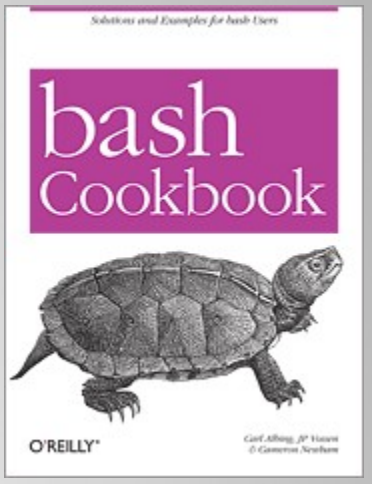
- bash recognizes special files
  - not part of file system
  - host is hostname or ip address
  - port is port number or service name
  - bash opens a connection to the socket
  - now compiled by default on ubuntu
  - or use `--enable-net-redirections` on compile

`/dev/tcp/`*host*`/`*port*

`/dev/udp/`*host*`/`*port*

bash - summary

- command line has many uses
- bash – for commands and scripts
- bash – a full-blown language
- many new, enhanced features
  - productivity, precision, flexibility
- a tool worth knowing well



bash info

- www.bashcookbook.com
  - many bash documents all in 1 place
- www.gnu.org/software/bash
- www.cygwin.com