

(1) cut from end

```
A="../usr/hallo.txt"      B=${A%/*}      "../usr"
                          B=${A%%/*}      ".."
```

(2) cut from begin

```
A="../usr/hallo.txt"      B=${A#*/}      "usr/hallo.txt"
                          B=${A##*/}      "hallo.txt"
```

(3) replace

```
A=123123123              B=${A/3/X}      "12X123123"
                          B=${A//3/X}      "12X12X12X"
                          B=${A/3}        "12123123"
                          B=${A//3}       "121212"

A=abcdef                 B=${A^}         "abcdef"      bash4
                          B=${A^^}         "ABCDEF"
A=ABCDEF                 B=${A,}         "aBCDEF"      bash4
                          B=${A,,}         "abcdef"
```

(4) cut out

```
A=abcdefghi              B=${A:2:3}      "cde"
```

(5) arithmetics

```
A=abcdefghi              B=${A: ${#A}-3 :3}  "cde"
                          B=${A:0: ${#A}-3 }  "abc"
```

(6) arrays

```
A[1]="h1";      A[2]="h2";      N=1;      A[${N]}="h3"

      number of elements      ${#A[@]}      3

${A[1]}      "h1"
${A[${N}]}      "h1"
${A[${N}]: ${#A[${N}]}-1 :1}      "1"
```

(7) default values

```
      # if A is...

${A:=/etc}      # ..not set, set it to "/etc"

${A:-/etc}      # ..not set, return "/etc"

${A:+/etc}      # ..not empty, return "/etc"

${A:? "A is not set"}      # ..empty or unset,
      # exit script with message

unset A      ${A:? "A not set"}      "", exit(1)

unset A      A=${A:=/etc}      "/etc"

      ${A:-/dev/null}      "/etc"

      ${A:+/usr}      "/usr"

      ${A:? "A not set"}      "/etc"

      ${A:+/usr}      ""
```

(8) process variable values

```
echo "${A}" | sed 's/l/XX/g'      # creating a pipe necessary

sed 's/l/XX/g' <<< ${A}      # this is better
```

(9) read one line into several variables:

```
while IFS=: read a b c d e f g ; do
    echo $a $b $c $d $e $f $g
done < /etc/passwd
```

..and read from file into an array:

```
while read -a A ; do
    for i in $(seq 0 ${#A[@]}) ; do
        echo -n "${A[$i]} "
    done
    echo ""
done < /etc/fstab
```

(10) avoid the evil read-subshell-problem

```
C=0
ls -la | {
    while read a ; do
        C=$((C+1))
    done
    echo ${C}
}

C=0
while read a ; do
    C=$((C+1))
done <<(ls -la)
echo ${C}
```

```
C=0
while read a ; do
    C=$((C+1))
done <<(ls -la)
echo ${C}
```

...and a function with two exits...

```
function hurbel {
    echo "entry[${*}] "
    echo "${*}" | while read a ; do
        echo "exit1[${a}] "
    done
    return 0
done
echo "exit2[${*}] "
return 1
}
```

(11) redirect input and output

```
#!/bin/bash
exec 10>&1      ; exec 11>&2  # save STDOUT+STDERR in FDs 10+11
exec 1>file.log ; exec 2>&1  # redir STDERR+STDERR to logfile
....
echo "hallo" 1>&10          # echo to default STDOUT
....
exec 1>&10      ; exec 2>&11  # restore default STDOUT+STDERR
exec 10>&-      ; exec 11>&-  # close unused FDs
```

..and read from a file:

```
declare -a A
typeset -i C=0
while IFS=: read a ; do
    C=C+1
    A[${C}]=${a}
    echo ${A[${C}]}
done < file.txt
echo "elements: ${#A[@]}"

declare -a A
C=0
exec 10<file.txt
while IFS=: read a <&10; do
    ((C++))
    A[${C}]=${a}
    echo ${A[${C}]}
done
echo "elements: ${#A[@]}"
exec 10>&-
```

(12) file tests

-b <filename>	block special file
-c <filename>	special character file
-d <directoryname>	directory exists
-e <filename>	file exists
-f <filename>	regular file exists
-G <filename>	file exists + owned by effective GID
-g <filename>	file exists + set-group-id
-k <filename>	sticky bit
-L <filename>	symlink
-O <filename>	file exists + owned by effective UID
-r <filename>	file is readable
-S <filename>	file is socket
-s <filename>	file is nonzero size
-u <filename>	file set-user-id bit
-w <filename>	file is writable
-x <filename>	file is executable

(13) string tests

-n <string>	non empty
-z <string>	empty

(15) convert binary/hexadecimal/octal

```
typeset -i BIN=1000 OCT=1000 HEX=1000 DEC
DEC=2#${BIN} ; echo "${DEC}"
DEC=8#${OCT} ; echo "${DEC}"
DEC=16#${HEX} ; echo "${DEC}"
```

(16) user input

```
echo -e "prompt: \c " ; read A
echo -n "prompt: " ; read A
```

(17) trap signals

```
TD=$(mktemp -d /tmp/XXXXXXXXXX)
T1=${TD}/tf1.tmp
trap "rm -Rf ${TD}" 0 1 2 3 12 13 15
...
exit 0
```

(18) two replacements for grep:

#---- as seen in c't 3/2009 ----

```
function __grep1 {
    exec 3<$2
    while read -u 3 a ; do
        [ -n "${a}" -a -z "${a/*${1}*/}" ] && echo "${a}"
    done
}
```

#---- Elbrands own version -----

```
function __grep2 {
    local R=1
    while read a ; do
        [ ! "${a}" = "" -a "${a/*${1}*/}" = "" ] && \
            { echo "${a}"; R=0; }
    done < "${2}"
    return ${R}
}
```

(19) statements

```
for A in $(seq 1 10) ; do echo "${A}" ; done

for ((A=0 ; A<10 ; A++)) ; do echo "${A}" ; done

while [ 1 -lt 2 ] ; do date ; done

until [ 1 -gt 2 ] ; do date ; done

select A in quit $(ls -1) ; do
    case $A in
        quit) break ;;
        *) echo $A ;;
    esac
done

function tester {
    date
    return 0
}

case ${A} in
    1) echo "1" ;;
    2*) echo "2" ;;& # make the next test (bash4)
    21) echo "21" ;;&
    22a) echo "22a" ;& # evaluate the next line (bash4)
    22x) echo "22x" ;;
esac
```

(20) here documents

(the awk is only to show the mode of operation)

```
cat <<EOF | awk '{print}'
1
2
EOF
```

you can indent the text. The operator '<<->' ignores leading TABs and the source code is formatted well

```
cat <<-EOF | awk '{print}'
<TAB>1
<TAB>2
EOF
```

(21) associative arrays (bash4)

```
declare -A X

X[k1]="v1"
X[k2]="v2"

echo "${!X[@]}" # list of indices
echo "${X[@]}" # list of values

for i in ${!X[@]} ; do
    echo "${X[$i]}"
done
```

(22) builtin: mapfile (bash4)

```
mapfile -n 0 lines < /etc/passwd

for i in $(seq 1 ${#lines}) ; do
    echo ${lines[$i]}
done
```

(23) callback function (bash4)

```
function command_not_found_handle {
    echo "command not found [${1}]"
}
```

(24) brace expansion

```
echo {05..10}      5 6 7 8 9 10
echo {05..10}      05 06 07 08 09 10    (bash4)
```

(25) Pattern matching

```
export LC_COLLATE=C ; export LC_ALL=C
```

```
ls -la [a-c]
```

```
ls -la [[:CLASS:]]
```

```
<CLASS>: { alnum | alpha | ascii | blank | cntrl | digit | graph | \
          lower | print | punct | space | upper | word | xdigit }
```

(26) avoid the evil while-ssh-problem

```
{ echo node01 ; echo node02 ; echo node03 ; } > f1.txt
```

a construction like this leaves the loop after one iteration

```
while read a ; do
    ssh ${a} uname -a
done < f1.txt
```

because the ssh reads from STDIN! - Try "-n" instead:

```
while read a ; do
    ssh -n -o BatchMode=yes ${a} uname -a
done < f1.txt
```

or, as a general solution:

```
while read a ; do
    : | ssh ${a} uname -a
done < f1.txt
```