Bash Scripting Cheatsheet

brought to you by the

BEND HACKERS GUILD

http://bend.hackersguild.us/

2017-08-08

BEND HACKERS GUILD **Bash Scripting Cheatsheet**

Note: *This is not a comprehensive reference by any stretch of the imagination!*

Contents

How to Read This

For blocks that begin with the $ bash shell prompt, bold represents user input. Bold is otherwise used to draw attention to a particular command.

Hello, World

Create a file `hello.sh` (`sh` extension is a convention, not required) containing:

```
#!/bin/bash
echo "Hello, world!"
```

Set permissions to execute:

```
$ chmod u+x hello.sh
```

Execute it:

```
$ ./hello.sh
Hello, World!
```

"Shebang"

The **`#!/bin/bash`** line tells the OS which program to execute this script with. (`#` is "hash" and `!` is "bang" for "shebang".)

Without this line, you'd have to run the script with the explicit interpreter:

```
$ bash hello.sh
```

Getting Help

View the Bash manual page:

```
man bash
```

View help for Bash built-in functions:

```
help test
help cd
help exit     (etc.)
```

Debugging

Run with tracing (each line is outputted before it executes):

```
$ bash -x hello.sh
```

You can activate tracing within a script at runtime:

```
#!/bin/sh
set -x  # turn tracing on
echo "Hello, world!"
set +x  # turn tracing off
```

Trace lines are normally preceded by +. You can change this in the PS4 variable.

You can also `trap` the DEBUG pseudosignal and run arbitrary code every line. See "Traps", below.

Or a third-party debugger might be useful. http://bashdb.sourceforge.net/

Exiting a shell or script

Use the exit call:

```
exit    # Exit successfully

exit 0  # Same as "exit"

exit 1  # Exit with error code 1

exit 5  # Exit with error code 5
```

If returning from a function, use `return` in place of `exit`. See Functions, below.

Semicolons

Semicolons separate commands and allow you to put multiple commands on the same line:

```
$ echo Goats; echo Detected
Goats
Detected
```

Colons

Colons are like comments, except they perform variable expansions and redirections. Nothing else is done. Space after the colon is required. Useful in the math section, below.

```
$ x=2
$ : $((++x))
$ echo $x
3
```

Comments

Any characters after an otherwise unused **#** are comments.

```
echo "Hello!"   # This prints "Hello!"
```

Variables

Referred to as *parameters* in the `man` page. All variables are parameters; not all parameters are variables.

Set a Variable

Set a variable to a value (***no spaces around =***):

```
a=hello
b=2
magic="The Magic Words are Squeamish Ossifrage"
```

Set a variable from the output of a command

Set a so it holds the first line of the file `foo.txt`:

```
a=$(head -1 foo.txt)
a=`head -1 foo.txt`   # equivalent, out of fashion
```

Get a Variable

Get a value from a variable with **$** prefix:

```
echo "Current magic is: $magic"
c=$b
```

Use `{}` around the name if you need to keep it separated from subsequent characters.

```
$ x=39
$ echo "He's in ${x}th place!"
He's in 39th place!
```

Append to a variable

```
$ x=abc
$ x+=def
$ echo $x
abcdef
```

Export a Variable

This causes a variable to be exported to subprocesses. If you don't do this, processes launched from this shell will not see the variable (which might or might not be what you want).

```
a=foo
export a
```

or

```
export a=foo1  # all on one line
```

Unset a Variable

If you want to remove a variable, you can

```
unset foo  # remove variable foo
```

Null variables

If a variable is set but has no value (or is empty string), it's said to be *null*. If you've `unset` it, it's said to be *unset*.

```
$ a=    # set a to null
$ b=""  # set b to null
```

Special Variables and Parameters

The shell sets a number of variables for you. Some of these you can write to, as well.

`$?`	Exit status of last command
`$#`	Number of command-line argument to the current script
`$0`	Name of currently-executing script
`$1`, `$2` ... `$9`	Command-line arguments (see shift, below)
`$*`	All command line arguments as a single string
`$@`	All command line arguments as separate strings
`$_`	The last argument to the last command executed
`$$`	The Process ID of the current process
`$!`	The Process ID of the last background process created
`$IFS`	Internal Field Separator; see IFS section below
`$PWD`	Current working directory
`$RANDOM`	A random number between 0 and 32767
`$FUNCNAME`	Name of currently-executing function
`$LINENO`	Current line number of the currently-executing script
`$HOSTNAME`	Name of host
`$GROUPS`	Groups to which the current user belongs
`$SECONDS`	Number of seconds since this shell was created

`$UID`	User ID of the current user
`$COLUMNS`	Screen columns
`$LINES`	Screen lines (rows)
`$PS1`	Main command-line prompt
`$PS3`	Prompt for the `select` statement
`$PS4`	Precedes lines of tracing output when debugging

Single Quotes, Double Quotes, and Escape

Both bind multiple words into a single argument.

Double Quotes

Double quotes still interpret special characters within:

```
$ echo "The value of a is: $a"
The value of a is: hello

$ echo "Files that start with foo: foo*"
Files that start with foo: foo foobar foobaz
```

Single Quotes

Single quotes do not interpret special characters:

```
$ echo 'foo* $magic'
foo* $magic
```

Escape

Escaping a special character with `\` removes its special meaning:

```
$ echo foo\* \$magic \\
foo* $magic \
```

Problems with Spaces in Filenames, etc.

See the `IFS` section.

Exit values

Zero (`0`) means "success" or "true", non-zero (often `1`) means "failure" or "false". Yes, it's backward from what you'd expect.

Exit values from the previously-executed command are found in the variable **`$?`**.

The **`true`** command always exits with value 0. The **`false`** command always exits with value 1.

Conditionals

[is a command that returns success (0) if a given true expression, and failure (1) if given a false expression. A closing] is required. ***Spaces between all arguments are required.***

This is a synonym for the infrequently-used `test` command.

```
[ 1 -eq 2 ]    # evaluates to failure

[ 1 -eq 1 ]    # evaluates to success

test 1 -eq 1   # evaluates to success, same as above
```

Example

```
$ [ 1 -eq 2 ]
$ echo $?
1     (failure, false)
```

Example tests

`[ 1 -eq 2 ]`	Test two integers for equality
`[ 1 -ne 2 ]`	Test two integers for inequality
`[ "foo" = "bar" ]`	Test two strings for equality
`[ "foo" != "bar" ]`	Test two strings for inequality
`[ -f "foobar" ]`	Test if file foobar exists and is a regular file
`[ ! 1 -le 2 ]`	True if 1 is **not** less than or equal to 2

Arithmetic tests

`-eq -ne -gt -lt -le -ge`

String tests

`= != < > -z` *(empty)* `-n` *(not empty)*

Boolean Logic

`[ $x -eq 2 -a $y -eq 3 ]`	True if x is 2 AND y is 3
`[ $x -eq 2 -o $y -eq 3 ]`	True if x is 2 OR y is 3

You can also use the shell's logical operators outside the calls to `test`. **&&** is AND and **||** is OR.

```
$ [ 1 -eq 3 ] && echo "Goats"
$ [ 1 -eq 1 ] && echo "Goats"
Goats
```

```
$ [ 1 -eq 1 ] || [ 1 -eq 3 ]
$ echo $?
0    (success, true)
```

List of All Available Tests

```
$ help test
```

Common String Comparison Idiom

If a string or variable is empty, string comparisons will fail:

```
$ foo=""
$ [ $foo = "" ]
bash: [: =: unary operator expected
```

This is worked around by padding the string with a non-empty value:

```
$ foo=""
$ [ x$foo = "x" ]  # expands to [ x = x ]
$ echo $?
0    (success, true)
```

Bash-specific Tests

[works in most (all?) shells, but Bash offers **[[** that supports regular expressions and other features. Use [when possible to be most portable.

Use ***no quotes*** on the regex (**=~**):

```
$ [[ "Beeeeeeeej" =~ Be+j ]] && echo "Beej found"
Beej found
```

See **help [[** for more.

if conditionals

An **if** statement evaluates the command up to the semicolon (**;**) and executes the statement body on success. The statement body is terminated with the backward variant **fi**.

```
if [ $n -eq 2 ]; then
    echo "Hey, n is two!"
fi

if [ $n -eq 2 ]   # Same as above, without the semicolon
then
```

```
    echo "Hey, n is two!"
fi

if grep -q clean Makefile; then  # -q to suppress grep output
    echo "Found the word \"clean\" in the Makefile"
fi

if [ -x /usr/bin/convert ]; then
    echo "Found the convert program in /usr/bin"
fi

if [ $x -eq 2 ] && [ $y -eq 3 ]; then
    echo "You sank my Battleship!"
    echo "...You bastard!"
fi
```

Else, Elif (else-if)

```
if [ $n -eq 2 ]; then
    echo "N is 2"
else
    echo "N is something else"
fi

if [ $n -eq 2 ]; then
    echo "N is 2"
elif
    echo "N is something else"
fi
```

If-elif-else Semicolon Placement

If you're into one-liners, this contrived example shows where the semicolons have to go:

```
if true; then a=2; elif false; then a=3; else a=4; fi
```

`for` loops

Repeatedly set a variable to a list of items.

```
for a in 1 2 3; do
    echo "a is now " $a
done
```

output:

```
a is now 1
a is now 2
a is now 3
```

Other examples

```
# Find and process directories
for a in $(find . -type d); do
     echo "Found directory " $a
done

# Process command line arguments
for a in $*; do
     echo "Command line argument: " $a
done
```

`while` loops

Repeat while a condition is true.

```
# Repeat while x != 99
while [ $x -ne 99 ]; do
    echo "x still isn't 99"
done

# Wait until goatfile.txt does not contain the word GOATS
while grep -q GOATS goatfile.txt; do
    sleep 5
done

# Repeat forever (true is a Unix command that always has exit
code 0)
while true; do
    echo "Take one down, pass it around..."
done

# Never enter this loop (false is a Unix command that always has
# exit code 1)
while false; do
    echo "You'll never see this message."
done

# Print 1 through 9
x=0
while [ $((++x)) -lt 10 ]; do  # See Math section
```

```
    echo $x
done
```

`until` loop

Logical opposite of a `while` loop.

```
# Repeat until x == 99
until [ x -eq 99 ]; do
    echo 'x *still* isn't 99!'
done
```

`case` statement

Choose code path based on value. AKA "switch" in other languages. The first matching pattern is used.

Match patterns use filename-style globbing expressions.

```
case "$resp" in
yes)
    echo "The user said yes!"
    ;;   # "break"
no)
    echo "The user said no!"
    ;;
*)
    echo "The user said something else"  # default case
    ;;
esac  # end of the case statement
```

Case pattern matching

```
case $x in
b*)
    echo "x starts with \"b\""
    ;;
*z)
    echo "x ends with \"z\""
    ;;
[4-8]*[dgh])
    echo "x starts with anything between 4 and 8 inclusive,"
    echo "and x ends with d, g, or h."
    ;;
```

```
    g??ts)
        echo "x starts with g, then any two letters, and ends with
    ts."
        ;;
    *)
        echo "x is any sequence of characters."
        ;;
    esac
```

select

Allows the user to select from a given list. Repeats until a **break** statement.

Prompt can be set in the **PS3** variable. (Default prompt is #**?**.)

```
    PS3="Choose: "

    select food in Beets Bagels Bacon Exit; do
        if [ "x$food" = xExit ]; then
            break
        else
            echo "You chose $food... wisely."
        fi
    done
```

```
    1) Beets
    2) Bagels
    3) Bacon
    4) Exit
    Choose: 3
    You chose Bacon... wisely.
    Choose: 2
    You chose Bagels... wisely.
    Choose: 1
    You chose Beets... wisely.
    Choose: 4
    $
```

Can also use path expansion or variable/array expansion:

```
    select file in *; do
        echo "$file looks like a fine file to me."
    done
```

$', Escaped Character Expansion

If you want a newline, tab, backspace, etc., you'll need to use this form.

`$'\t'`	Tab character
`$'\n'`	Newline
`$'\r'`	Carriage return
`$'\e'`	Escape
`$'\b'`	Backspace
`$'\xXX'`	Hex character (up to 2 digits)
`$'\uUUUU'`	Unicode character (up to 4 digits)
`$'foobar\n'`	"foobar" followed by a newline

$IFS, Internal Field Separator

This controls how words are split up during variable expansion and during user input, below.

```
IFS=:
v=1:2:3   # Since IFS works on var expansion, must be in a var

for a in $v; do
    echo "a is now " $a
done
```

output:

```
a is now 1
a is now 2
a is now 3
```

It's common to set IFS to comma for CSV processing, or to newline for processing input containing spaces.

```
IFS=$'\n'

v="This is a multi-line string
This is the second line
And this is the third line"

for a in $v; do
    echo "line: $a"
done
```

output:

```
line: This is a multi-line string
line: This is the second line
line: And this is the third line
```

Command Line Arguments

The name of the currently-executing script is in **$0.** The following examples assume the script has been launched with three arguments: `1 2 3`.

```
echo "This script is " $0
echo "This script (path stripped) is " $(basename $0)
```

The rest of the arguments are in parameters **$1** through **$9**.

For easy parsing (and more than 9 arguments), see `shift`, below.

```
echo "Argument 1 is $1"
echo "Argument 2 is $2"
```

$* and $@

These expand to all arguments. **"$*"** is all arguments as a single value, and **"$@"** is all arguments as separate values.

Use quotes around the variables for proper grouping after expansion.

```
for a in "$*"; do
    echo "arg: $a"
done

arg: 1 2 3
```

```
for a in "$@"; do
    echo "arg: $a"
done

arg: 1
arg: 2
arg: 3
```

shift

You can shift all arguments "left" into their previous $*n* slots, e.g. $2 is shifted into $1, $3 is shifted into $2, etc. The number of arguments in $# is decremented.

```
while [ $# -gt 0 ]; do
    echo "arg: $1"
    shift
done
```

```
arg: 1
arg: 2
arg: 3
```

functions

Functions are invoked just as if they were external programs. They also receive command line arguments in the same way.

Use* `return` *instead of* `exit`*. `exit` still exits the entire script like normal.

Declare:

```
function freakout {
    echo "AAaaaaagghh!"
}
```

then:

```
$ freakout
AAaaaaagghh!
```

`return` (instead of `exit`!)

```
function prefix {
    if [ $# -ne 1 ]; then
        echo "Missing argument to prefix function"
        return 1
    fi

    echo "This is the prefix. $1"
}

prefix
echo $?
prefix goats
echo $?
```

output:

```
Missing argument to prefix function
1
This is the prefix. goats
0
```

Local variables

Variables are all global to the script unless explicitly declared local in a function.

```
a=2   # globals
b=3

function localdemo {
    local b    # local "b" hides the global "b"

    a=33       # modifies the global "a"
    b=44       # modifies the local "b"

    echo "function: a is now $a"
    echo "function: b is now $b (local)"
}

echo "global: a is $a"
echo "global: b is $b"

localdemo

echo "global: a is now $a"
echo "global: b is still $b"
```

output:

```
global: a is 2
global: b is 3
function: a is now 33
function: b is now 44 (local)
global: a is now 33
global: b is still 3
```

Math with $((...))

Arithmetic can be computed with this expansion.

```
$ echo $((1 + 3))
4

$ x=10
$ echo $((5 * $x))
50
```

```
$ echo $(( $(( 5 + 3 )) * $((1 + 4)) ))
40
```

Basic math

`- +`	unary minus and plus
`**`	exponentiation
`* / %`	multiplication, division, remainder
`+ -`	addition, subtraction
`<< >>`	left and right bitwise shifts
`~`	bitwise negation
`&`	bitwise AND
`^`	bitwise exclusive OR
`|`	bitwise OR

Math that modifies variables

`v++ v--`	variable post-increment and post-decrement
`++v --v`	variable pre-increment and pre-decrement
`= *= /= %= += -=`	assignment
`<<= >>= &= ^= |=`	more assignment

```
$ x=4
$ echo "After this, x will be $((x += 4))"
After this, x will be 8
$ echo $x
8
```

It's probably more common to do assignments this way, however:

```
$ x=$(($x + 4))   # add 4 to x
```

Or

```
x=0
while [ $((x++)) -lt 10]; do
    echo x is $x
done
```

Use the colon operator (`:`) to perform the expansion but do nothing else (otherwise the shell tries to execute the result as a command):

```
$ : $((++x))      # just increment x, that's all
```

Boolean Logic

For boolean logic in `$((...))`, `0` means false and non-zero means true, ***the opposite meaning from the exit values in the shell***.

`!`	logical negation
`&&`	logical AND
`||`	logical OR
`<= >= < >`	comparison
`== !=`	equality and inequality

```
$ x=2
$ y=3
$ echo $(($x == 2 && $y == 999))
0   (false)
```

Ternary Operator `?:`

If the first subexpression is true, evaluates to the second, otherwise it evaluates to the third. The second two subexpressions must be numeric.

condition ? *if-true* : *if-false*

```
$ x=4
$ echo $(($x == 99? 6: 8))
8
```

Floating Point Math

If you have the Unix `bc` tool installed (usually standard), you can make a function that uses it for floating point math. `man bc` for all the functions you have at your disposal.

```
function math {
    echo "$*" | BC_LINE_LENGTH=0 bc -l  # bc minus ell
}

a=2.8
b=3.8
x=$(math $a / $b)

echo $x   # prints ".736842105263157894 73"

echo $(math sqrt\($x\))    # sqrt of 2.8
1.673320053068151095

echo $(math $a \< 5)       # is 2.8 < 5?
```

```
1

echo $(math $a \> 5)       # is 2.8 > 5?
0

echo $(math 'scale=50; 4 * a(1)')   # 4 times arctangent of 1, or
π
3.14159265358979323846264338327950288419716939937508
```

Math with `expr`

`expr` is another common Unix utility that works similarly to the above. Spaces around arguments are required. `$((...))` is faster since it's built into Bash.

```
$ x=6
$ expr $x + 3
9
$ y=$(expr $x + 10)  # y is assigned 16
```

Advanced Parameter Substitutions and Expansions

`${parameter:-word}`	If *param* is unset or null, substitute *word*

```
$ a=""
$ b=20
$ echo ${a:-"a is null!"}
a is null!
$ echo ${b:-"b is null!"}
20
```

`${parameter:=word}`	Same as above, but also assigns *word* into *parameter*
`${parameter:?word}`	Display error if *parameter* null or unset
`${parameter:+word}`	Opposite behavior of `${parameter:-word}`
`${!prefix*}`	Show names of variables that start with prefix
`"${!prefix@}"`	Same as above, except each var is a separate word

String Manipulation

`${#parameter}`	Parameter length
`${param:offset}`	
`${param:offset:length}`	Extract substrings from *param*, negative offset from end

The following use filename globbing patterns for matching *(i.e. * and ?)*

`${param#pattern}`	Remove prefix matching pattern from param

${*param##pattern*}	Same as above, greedy
${*param%pattern*}	Remove suffix matching pattern from param
${*param%%pattern*}	Same as above, greedy

```
$ b=foobar
$ echo ${b%bar}
foo

$ a='abc;def;ghi;jkl'

$ echo ${a#*;}      # Remove everything through the first ;
def;ghi;jkl

$ echo ${a##*;}     # Remove everything through the last ;
jkl

$ echo ${a%;*}      # Remove everything from the last ;
abc;def;ghi

$ echo ${a%%;*}     # Remove everything from the first ;
abc
```

${*param*/*pattern*/*string*}	Perform pattern substitution or deletion

```
v=abab
```

${v/b/X}	Substitute first: aXab
${v/#a/X}	Substitute at start of line: Xbab
${v//a/X}	Substitute all: XbXb
${v/%b/X}	Substitute at end of line: abaX
${v/b}	Delete first: aab
${v/#a}	Delete at start of line: bab
${v//a}	Delete all: bb
${v/%b}	Delete at end of line: aba

${*parameter*^}	Convert first character to uppercase
${*parameter*^^}	Convert all characters to uppercase
${*parameter*,}	Convert first character to lowercase
${*parameter*,,}	Convert all characters to lowercase
${*parameter*^*pattern*}	Same as above except with pattern matching
${*parameter*^^*pattern*}	Same as above except with pattern matching
${*parameter*,*pattern*}	Same as above except with pattern matching
${*parameter*,,*pattern*}	Same as above except with pattern matching

Extra Special Expansions

`${`*`parameter`*`@Q}`	Show value quotable as input
`${`*`parameter`*`@E}`	Show value with backslash escape sequences expanded
`${`*`parameter`*`@P}`	Expand value as if it were a prompt
`${`*`parameter`*`@A}`	Show assignment that would create this parameter
`${`*`parameter`*`@a}`	Show parameter flag values

@P is particularly powerful, since the prompt has the capability of outputting all kinds of specialized information.

```
$ v='Time \t, user \u, hostname \H'
$ echo ${v@P}
Time 14:30:52, user beej, hostname goatee
```

@E can expand other special escapes:

```
$ v='foo\n\tbar'   # \n newline, \t tab
$ echo $v
foo\n\tbar
$ echo "${v@E}"
foo
        bar
```

Range Expansions

```
{1..3}              1 2 3
{04..12}            04 05 06 07 08 09 10 11 12
{0..10..2}          0 2 4 6 8 10
```

Arrays

One-dimensional, indexed and associative.

Indexed Arrays

`declare -a` *`param`*	Declare an indexed array variable *param*
`local -a` *`param`*	Declare an indexed array local variable *param*
`read -a` *`param`*	Read user input into an array variable *param*
`param``=(abc def ghi)`	Declare an indexed array with three elements
`${a[12]}`	Access element 12 of array *a*

`param+=(jkl mno)`	Append elements onto array
`param+=(${param2[*]})`	Append array's elements onto array
`${param[*]:5}`	Expand the rest of the elements starting with the 5th
`${param[*]:8:2}`	Expand 2 elements starting with the 8th
`${param[*]:-5:3}`	Expand 3 elements starting with the 5th-from-the-last

Associative Arrays

`declare -A param`	Declare an associative array variable *param*
`local -A param`	Declare an associative array local variable *param*
`param=([a]=Z [b]=Y [c]=X)`	Declare an associative array with three elements ***Must use* `declare -A` *or* `local -A` *before this!***
`param+=([d]=W [e]=V)`	Append elements onto array

Common Functionality

`param[2]="Hello!"`	Assign into *param*[2]
`${param[2]}`	Get value *param*[2]
`${param[-2]}`	Get value of second-to-last element in array
`${#param[2]}`	Get length of value *param*[2]
`${#param[*]}`	Return the number of elements in the array
`${#param[@]}`	Return the number of elements in the array, also
`"${param[*]}"`	Expand to a single word with all elements of array
`"${param[@]}"`	Expand to separate words all elements of array
`"${!param[*]}"`	Expand to a single word with all keys of array
`"${!param[@]}"`	Expand to separate words all keys of array
`unset param`	Destroy an array
`unset 'param[2]'`	Destroy *param*[2] in particular

printf

Venerable function for formatted output. First argument is the *format string*. Subsequent arguments are included in their respective **%** substitution points. End in **\n** for a newline.

```
printf "Hello, world!\n"
printf "x holds an integer value: %d\n" $x
printf "y holds a floating point value: %f\n" $y
```

```
printf "z holds a string: %s\n" "$z"
printf "all: x=%d, y=%f, z=%s\n" $x $y "$z"
```

`printf "%8d" 3490`	Field width, prints " 3490"
`printf "%-8d" 3490`	Field width, prints "3490 "
`printf "%08d" 3490`	Field width, prints "00003490"
`printf "%9.2f" 3.14159`	Field width and decimal places: " 3.14"

`man 3 printf` for details of the format string.

User input

`read` *param*	Read a line of user input into variable *param*
`read -a` *param*	Read a line of user input, split into array *param*
`IFS=, read -a` *param*	Read into array *param*, comma-separated
`read -d` *c param*	Read up to character *c* into *param* (instead of newline)
`read -e` *param*	Use readline (superior editing capabilities)
`read -n` *num param*	Read up to *num* character into *param*
`read -p` *prompt param*	Display prompt before reading
`read -r` *param*	Do not interpret backslash escapes during read
`read -s` *param*	Read silently; do not echo keyboard input
`read -t` *secs param*	Timeout after *secs* seconds on the read

File input

`read -u fd param`	Read from file descriptor `fd`

File I/O

Files are opened and closed with the `exec` built-in. (If no command is given, exec runs any given redirections in the current shell.)

The first 3 descriptors are generally already in use. `0` is *standard input*, `1` is *standard output*, and `2` is *standard error*. AKA *stdin*, *stdout*, *stderr*.

You can open files by explicit file descriptor number, or with a variable name in braces.

Reading a file

```
exec {fd}< foo.txt   # open file foo.txt for reading with var fd
line_count=1

# Loop through, reading and outputting lines
while read -u $fd line; do
    printf "Line %d: %s\n" $line_count "$line"
```

```
    : $((line_count++))
done

exec {fd}<&-   # close file descriptor fd for reading
```

Writing to a file

```
exec {fd}> foo.txt   # open file foo.txt for writing with var fd

# Write three lines to descriptor fd, see "Dup" section for more
printf "Hello!\n" >&$fd
printf "This is the second line.\n" >&$fd
printf "This is the third line.\n" >&$fd

exec {fd}>&-   # close file descriptor fd for writing
```

Using Numbers for File Descriptors directly

```
exec 3> foo.txt      # open file foo.txt for writing on file desc
3
printf "Hello!" >&3  # write Hello! to fd 3
exec 3>&-            # close file descriptor 3 for writing
```

Summary

If `fd` is in braces `{fd}`, then it refers to a variable named `fd`. Otherwise it should be a number.

`exec` *`fd< filename`*	open *filename* into descriptor *fd* for reading
`exec` *`fd> filename`*	open *filename* into descriptor *fd* for writing
`exec` *`fd<> filename`*	open *filename* into descriptor *fd* for reading and writing
`exec` *`fd`*`<&-`	close descriptor *fd* for reading
`exec` *`fd`*`>&-`	close descriptor *fd* for writing

Duplicating ("duping") file descriptors with >& and <&

This changes one file descriptor to another. Useful if you want a command that normally outputs on one file descriptor (e.g. standard output, 1) and you want it to go to another (e.g. an open file or standard error, 2).

`>&`*`fd2`*	Make all output normally to stdout go to *fd2* instead
`fd1``>&`*`fd2`*	Make all output normally to *fd1* go to *fd2* instead
`<&`*`fd2`*	Make all input normally from stdin come from *fd2* instead
`fd1``<&`*`fd2`*	Make all input normally from *fd1* come from *fd2* instead

`exec` *fd*`<&-` close descriptor *fd* for reading (dup into "closed")
`exec` *fd*`>&-` close descriptor *fd* for writing (dup into "closed")

```
echo "This goes to standard output"
echo "This goes to standard error" >&2
echo "This goes to standard error, too" 1>&2
echo "This goes to file descriptor 3" >&3
echo "This goes to some file descriptor in var fd" >&$fd
```

```
cat <&3     # give cat input from file descriptor 3
cat <&$fd   # give cat input from file desc in var fd
```

Here Documents and Here Words

Redirection that allows bulk writing to file descriptors and processes.

Here Documents

Send several lines of text to `cat` (***no space between << and marker!***):

```
x=2

cat <<_EOF_    # _EOF_ can be any unique marker
This is line 1
This is line 2 and x = $x
_EOF_
```

Send several lines of text to `file.txt`:

```
exec {fd}> file.txt # open file for writing

cat <<_EOF_ >&$fd   # redirect to fd
This is line 1
This is line 2
_EOF_

exec {fd}>&-        # close file for writing
```

Leading Tabs

If you put a minus between the << and the marker, leading tabs will be stripped (helpful for indentation within in the here-document):

```
cat <<-EOF
     ...
EOF
```

Here Strings

Like here documents, except just a string that has its variables expanded and a newline added:

```
x=2
cat <<<"Hello world! x is $x"
```

Subshells

Any commands wrapped in parentheses **( )** are run in a subshell. Open files are inherited by the subshell and `exported` variables are accessible.

Useful when you want to make temporary changes to the program state (changing directory, setting variables) that you want to revert when the subshell exits.

```
pwd                             # says we're in /home/foo
( cd /var/log; tail syslog )    # change directory in a subshell
pwd                             # still in /home/foo
```

Redirection with subshells

You can redirect the entire output:

```
( head foo.txt; tail bar.txt; cat baz.txt ) > output.txt
```

eval

Evaluate a string as if it were entered on the command line. Common use is to indirectly refer to variables.

Never blindly eval unsterilized user input!

```
foo=2
var=foo
eval echo \$$var    # expands to "echo $foo", prints 2
```

Sterilizing input

Use **printf** with the **%q** format specifier to escape all shell special characters.

```
$ x='Hello world; rm -rf *'         # badness, do not eval!!
$ eval echo "User entered: " $x     # BAD! NOOOOOooooooo!!
User entered: Hello world           ← But all your files are gone!

$ y=$(printf "%q" "$x")             # %q escapes all special shell
chars
$ echo $y
Hello\ world\;\ rm\ -rf\ \*
```

```
$ eval echo "User entered: " $y
User entered: Hello world; rm -rf *  ← That's what we wanted to see
```

Background Tasks

Backgrounding

You can run tasks in the background by appending an ampersand:

```
$ sleep 20 &          (returns immediately)
[1] 10951             (job 1, process ID 10951)
$ echo $!             (PID of latest background process)
10951
```

The job and process is only printed in interactive shells, not in scripts.

Waiting

You can **wait** for tasks to complete:

`wait %1`	wait for job 1 to complete
`wait 10951`	wait for PID 10951 to complete
`wait $!`	wait for the latest background process to complete
`wait -n`	wait for any background process to complete

The exit status $? after wait is the same as the exit status of the waited-for process, or 127 if there is no process to wait on.

Example

```
sleep 6 &
read -p 'Enter something> ' a
wait -n
echo "Done reading and sleeping"
```

Piping loops

You can pipe loop output to other commands.

```
# Don't output the first line read from the keyboard
# and save the rest in foo.txt

while read -p 'prompt> ' a; do
    echo "read: $a"
done | tail -n +2 > foo.txt
```

Process Substitution

Treats commands as files, so a place where you'd put an input file or output file on the command line, you can put another command to be piped to or from, instead.

Compare the output of `man bash` and `man sh`:

```
$ diff <(man bash) <(man sh)    # diff needs two input files
```

Write a compressed version of output from `tee`:

```
$ ls -lR | tee >(gzip -c > ls_lr.txt.gz)
```

In the above example, `tee` normally expects a filename. We substitute a process that gets tee's output on `stdin`.

Traps

You can trap signals and call code (usually a function) when it happens.

`trap -l`	List all trappable signals
`trap -n`	Show currently-trapped signals
`trap` *`handler signal`*	Call *handler* on *signal*, e.g. SIGTERM, TERM, or 15
`trap -` *`signal`*	Stop trapping a *signal*
`trap` *`signal`*	Stop trapping a *signal*, also

Pseudo-signals

The **`EXIT`** trap handler gets called when the script exits.

```
function on_exit {
    echo "I'm finished."
}

trap on_exit EXIT
```

The **`ERR`** trap handler gets called on *some* command errors.

```
function on_error {
    echo "I detected an error!"
}

trap on_error ERR
```

The **RETURN** trap handler gets called when a function returns, or when a `source`'d script (the `.` command) finishes running.

The **DEBUG** trap handler gets called for every line of execution the trap is in effect. Here's a simple debugger that executes the next line when RETURN is pressed on a blank line. Otherwise the line is `eval`'d.

```
function debug {
    local _debug_src _debug_lineno _debug_line _debug_have_input
    local _debug_command

    _debug_src=${BASH_SOURCE[1]}
    _debug_lineno=${BASH_LINENO[0]}
    _debug_line=$(tail -n +$_debug_lineno $_debug_src | head -1)

    _debug_have_input=1

    while [ $_debug_have_input -gt 0 ]; do
        printf "\n%s\n" "$_debug_line"

        read -e -p "debug> " _debug_command

        if [ "x$_debug_command" != x ]; then
            eval "$_debug_command"
        else
            _debug_have_input=0
        fi
    done
}

v=2

echo "Line 1"
echo "Line 2, v is $v"

trap debug DEBUG   # turn debugging on

echo "Line 3, v is $v"
echo "Line 4, v is $v"

trap - DEBUG       # turn debugging off
```

```
echo "Line 5, v is $v"
echo "Line 6"
```

Example run, setting variable v on the fly:

```
Line 1
Line 2, v is 2

echo "Line 3, v is $v"
debug> v=4

echo "Line 3, v is $v"
debug> [RETURN]
Line 3, v is 4

echo "Line 4, v is $v"
debug>
```

Co-Processes

A coprocess runs in the background and sets up two file descriptors that can be used to write to and read from the coprocess.

Think of it like a process running as a service that you can write to and read from through the file descriptors.

You might also be able to control an interactive program with this.

Important: the command running the coprocess should be set to "unbuffered", see http://mywiki.wooledge.org/BashFAQ/009 . The `-u` in the `sed` example below does this.

```
# set up a coprocess that will capitalize all
# instances of the word "goat"
coproc mycp { sed -u 's/goat/GOAT/g' ;}

# send lines of data to the coprocess
echo "test with antelope" >&${mycp[1]}
echo "test with goat" >&${mycp[1]}

echo "Results:"

# read lines of data from the coprocess
read -ru ${mycp[0]} result
echo $result                # "test with antelope"
```

```
read -ru ${mycp[0]} result
echo $result                # "test with GOAT"

kill $mycp_PID  # end the coprocess
```